Model-Based Software Component Testing

Weiqun Zheng

B.Sc., M.Eng.

This thesis is presented for the degree of

Doctor of Philosophy

of

The University of Western Australia

School of Electrical, Electronic and Computer Engineering

Faculty of Engineering, Computing and Mathematics

The University of Western Australia

March 2012

To My Parents

To Rongrong and Feifei

Statement of Originality

The work presented in this thesis was undertaken entirely by the author's sole PhD research and is, to the best of the author's knowledge, original, except where due references or acknowledgments have been made in the text of this thesis.

The material contained in this thesis has not previously been submitted, in part or in full, for a degree at this or any other university.

Weiqun Zheng

School of Electrical, Electronic and Computer Engineering (EECE)
Faculty of Engineering, Computing and Mathematics (ECM)
The University of Western Australia (UWA)
March 2012

Abstract

Software component testing (SCT) is a proven software engineering approach to evaluating, improving and demonstrating component reliability and quality for producing trusted software components, which is critical to support the success of component-based software engineering. Model-based testing (MBT) of software components enables the utilisation of a consistent model-based approach and specification (e.g. UML models) for effective component development and testing. However, advancing from model-based development to MBT poses certain crucial challenging problems that remain unresolved and hamper the utilisation of SCT/MBT, and further research is thus required to address those problems to achieve the goal of desirable SCT/MBT effectiveness.

This thesis has comprehensively reviewed the important concepts, principles, characteristics and techniques of SCT/MBT in the literature to provide a solid foundation for this research and introduced a set of useful new concepts and definitions to form the first major part of the thesis's original contributions as follows:

1. In the research areas of software components and software component testing:
   (a) A new comprehensive taxonomy of software component characteristics
   (b) A new software component definition
   (c) A new definition of software component testing
   (d) A useful taxonomy of software component testing techniques
   (e) A practical taxonomy of component testability improvement approaches

2. In the research areas of model-based testing and UML-based testing:
   (a) A study of model-based tests
   (b) A new definition of model-based testing
   (c) A new test model definition
   (d) A new definition of UML-based testing
   (e) A core UML subset for SCT
   (f) A study and review of use case driven testing and scenario-based testing

The principal original contribution of this thesis is to introduce a novel hybrid SCT methodology, called *Model-Based Software Component Testing* (MBSCT), which consists of five major methodological components, a three-phase testing framework, six main methodological features and six core testing capabilities. In more detail:

(1) The *Model-Based Integrated SCT process* incorporates software component

development and testing into a unified UML-based software process as part of the software development lifecycle, which provides a useful process model for the entire MBSCT methodology. This process supports the use of a consistent UML-based approach and specification for systematically developing test models and model-based component tests.

(2) The *Scenario-Based Component Integration Testing technique* focuses testing priority on identifying and constructing appropriate test scenarios to exercise and examine crucial deliverable component functions with the associated operational use case scenarios (e.g. behavioural instances and integration scenarios). This technique specifically supports the development of scenario-based test models and scenario-based test cases for component integration testing that bridges component unit testing and component system testing.

(3) The *Test by Contract technique* introduces a new test contract notion as a key testing-support mechanism and a set of useful contract-based concepts and test contract criteria to improve component testability and bridge the identified "*test gaps*" in MBT. This technique provides a stepwise TbC working process and a contract-based fault detection and diagnosis method to facilitate test model construction, component test design and generation, component fault detection, diagnosis and localisation, which establishes the major technical foundation for component test evaluation.

(4) The *Test-Centric Remodeling strategy* provides a practical guide to assist test model construction and model-based test derivation by means of *test-centric model refinement*, *model-based testability improvement* and *test-centric model optimisation*. This strategy works collaboratively with the corresponding MBSCT methodological components.

(5) The *Component Test Mapping technique* is developed as a new mapping-based test derivation approach, and focuses on mapping and transforming testing-related model artefacts and associated test contracts into useful test data for generating target component test cases.

This thesis has undertaken a comprehensive validation and evaluation of the MBSCT methodology, which has demonstrated and confirmed that it is effective in achieving the required level of component correctness and quality. The methodology comparison has concluded that the MBSCT methodology has significant advantages over the most-cited representative SCT/MBT approaches reported in the literature. This thesis has achieved substantial and original contributions to the Software Engineering scholarly body of knowledge in terms of the substantial literature review of SCT/MBT and the comprehensive MBSCT methodology. The research results presented in this thesis should provide a solid foundation for further research into SCT/MBT, which can help to bring closer the ultimate goal of achieving effective model-based component testing and producing trusted quality software components.

Publications

This thesis is based on a number of the author's publications, including research papers and technical reports, which are part of the outcomes of the sole-authored research work during the PhD candidature.

Some of the main research results and original contributions of this thesis has been formally published in the following six research papers (book chapter, journal article, conference paper and presentation):

[1] Weiqun Zheng and Gary Bundell, "A UML-Based Methodology for Software Component Testing," *Proc. The 2007 International Conference on Software Engineering* (*ICSE 2007*), Hong Kong, 21–23 March 2007, pp. 1177–1182.

[Note: The conference program committee has nominated this paper for the Best Paper Award of ICSE 2007, and also recommended this paper for the edited book published by Springer. This paper was awarded a Certificate of Merit.]

[2] Weiqun Zheng and Gary Bundell, "Model-Based Software Component Testing: A UML-Based Approach," *Proc. 6th IEEE International Conference on Computer and Information Science* (*ICIS 2007*), Melbourne, Australia, 11–13 July 2007. IEEE Computer Society Press, 2007, pp. 891–898.

[3] Weiqun Zheng and Gary Bundell, "A Framework of UML-Based Software Component Testing," Book Chapter 40, in Oscar Castillo, Li Xu and Sio-Iong Ao (Eds.), *Current Trends in Intelligent Systems and Computer Engineering*, Lecture Notes in Electrical Engineering, vol. 6, pp. 575–597, Springer, May 2008.

[4] Weiqun Zheng and Gary Bundell, "Test by Contract for UML-Based Software Component Testing," *Proc. 2008 IEEE International Symposium on Computer Science and its Applications* (*CSA 2008*), Hobart, Australia, 13–15 Oct 2008. IEEE Computer Society Press, 2008, pp. 377–382.

[5] Weiqun Zheng and Gary Bundell, "Contract-Based Software Component Testing with UML Models," *International Journal of Software Engineering and Its Applications*, vol. 3, no. 1, pp. 83–102, January 2009.

[6] Weiqun Zheng, Gary Bundell and Terry Woodings, "UML-Based Software Component Testing," *2010 Symposium in association with the Software Engineering Forum on*

Progress in Software Testing, ITEE College, Engineers Australia, Perth, Australia, July 2010.

In addition, some of the main research results and original contributions of this thesis has been also published and presented in a number of technical reports produced during the course of this research. The following includes the technical reports most relevant to this thesis:

[1] Weiqun Zheng, "Software Component Testing and Certification – The Software Component Laboratory Project," Technical Report CIIPS_ISERG_TR–2006–01, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[2] Weiqun Zheng, "Towards a Standard Test Specification for Software Component Testing," Technical Report CIIPS_ISERG_TR–2006–02, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[3] Weiqun Zheng, "Model-Based Software Component Testing – An UML-Based Approach to Software Component Testing," Technical Report CIIPS_ISERG_TR–2006–03, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[4] Weiqun Zheng, "Component-Based Software Development with UML and RUP/UP – Case Study: Car Parking System," Technical Report CIIPS_ISERG_TR–2006–04, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[5] Weiqun Zheng, "Model-Based Software Component Testing: A Methodology in Practice," Technical Report CIIPS_ISERG_TR–2006–05, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[6] Weiqun Zheng, "Model-Based Software Component Testing – Case Study: Car Parking System," Technical Report CIIPS_ISERG_TR–2006–06, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[7] Weiqun Zheng, "Applying Test by Contract to Improve Software Component Testability," Technical Report CIIPS_ISERG_TR–2007–02, Centre for Intelligent

Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2007.

[8] Weiqun Zheng, "Model-Based Approaches: Models, Modeling and Testing," Technical Report CIIPS_ISERG_TR–2009–01, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2009.

[9] Weiqun Zheng, "Model-Based Software Component Testing – Case Study: Automated Teller Machine System," Technical Report CIIPS_ISERG_TR–2010–01, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2010.

Acknowledgements

Firstly, I would like to express my sincere appreciation and gratitude to my principal supervisor, Professor Gary Bundell, for his invaluable guidance, advice, encouragement, support, patience and friendship throughout my PhD candidature. He has made constructive research discussions with me, and insightful suggestions and review comments on the drafts and revisions of my research papers, technical reports and PhD thesis, even when he has had heavy administrative duty as the Head of EECE School, busy schedule of full-time teaching and research, or on his sabbatical leave. His guidance over the years has significantly improved my research and writing skills, and I have learned a lot from him at both academic research and personal levels. This PhD would not have been possible without his great supervision. I am very grateful to have him as my supervisor and friend.

I would like to thank my co-supervisor, Professor Thomas Bräunl, Director of CIIPS, for his generous support, encouragement and friendship, especially in the later stage of my PhD candidature.

I would like to thank Professor Terry Woodings for providing his valuable feedback on the final thesis revision.

I would like to thank Professor Brett Nener, the current Head of EECE School and Deputy Dean (International Relations) of ECM Faulty, for his generous support in the later stage of my PhD candidature.

I would like to thank the generous help and support from staff of CIIPS, EECE School and UWA Graduate Research and Scholarships Office. I thank Ms Linda Barbour for her kind assistance with administrative matters. Our staff are always vey helpful and supportive.

I would like to thank my friends, fellow postgraduate students and members from Information and Software Engineering Research Group (ISERG), High-Integrity Computer Systems Laboratory (HICSL), Centre for Intelligent Information Processing Systems (CIIPS), School of Electrical, Electronic and Computer Engineering (EECE), School of Computer Science and Software Engineering (CSSE), Faculty of Engineering, Computing and Mathematics (ECM), UWA. You know who you are and this is how we get on well together. Your friendship and support greatly help me to get through this journey.

Finally and most importantly, I would like to express my grateful appreciation to my parents, my family, especially my loving wife Rongrong and son Feifei. I dedicate this thesis to all of you. More than anyone else, my wife has shared my highs and lows, and especially has endured family hardship and daily housework, fits of my depression and bad temper, as well as long hours away from home. This long journey would not have been possible without your endless love, understanding, encouragement, support and companionship during this difficult period. Thank you.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

ATM	Automated Teller Machine
CBCTD	Contract-Based Component Test Design
CBFDD	Contract-Based Fault Detection and Diagnosis
CBS	Component-Based Software/System
CBSE	Component-Based Software Engineering
CDD	Component Descriptor Document
CfD	Contract for Diagnosability
CfT	Contract for Testability
CIT	Component Integration Testing
COTS	Commercial-Off-The-Shelf (component)
CPS	Car Parking System
CTC	Component Test Case
CTI	Component Testing Index
CTM	Component Test Mapping
CTS	Component Test Specification
CUT	Component Under Test
DbC	Design by Contract
DOTM	Design Object Test Model
DTD	Document Type Definition
ETC	External Test Contract
FDD	Fault Detection and Diagnosis
GUI	Graphical User Interface
IBT	Implementation-Based Testing
IEEE	The Institute of Electrical and Electronics Engineers, USA
ITC	Internal Test Contract
MBD	Model-Based Design/Development
MBSCD	Model-Based Software Component Development
MBSCT	Model-Based Software Component Testing
MBT	Model-Based Testing
MDA	Model-Driven Architecture

MDD	Model-Driven Development
MDE	Model-Driven Engineering
OCL	Object Constraint Language
ODM	Object Design Model
OMG	Object Management Group, USA
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOT	Object-Oriented Testing
PAL	Parking Access Lane
PIN	Personal Identification Number
RSD	Result Set Document
SBT	Specification-Based Testing
SCD	Software Component Design/Development
SCI	Software Component Integration
SCL	Software Component Laboratory
SCT	Software Component Testing
SDLC	Software/System Development Life Cycle
SUT	Software/System Under Test
TbC	Test by Contract
TCR	Test-Centric Remodeling
TC	Test Contract
TG	Test Group
TM	Test Mapping
TO	Test Operation
TPV	Test Pattern Verifier
TS	Test Sequence/Scenario
TSD	Test Specification Document
TUC	Test Use Case
UBT	UML-Based Testing
UCM	Use Case Model
UCTM	Use Case Test Model
UML	Unified Modeling Language

UML–SCT	A core UML subset for SCT
UP	Unified Process
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1
Introduction

This chapter presents an overall introduction to this thesis. For the purpose of this research, Section 1.1 discusses the main research problems and challenges of primary interest in the research areas of software component testing and model-based testing. Section 1.2 describes the primary motivations and objectives of this research. Then, Section 1.3 presents an overview of the original contributions of this thesis. Section 1.4 outlines the structure of this thesis.

1.1 Research Problems and Challenges

Since the term of "software components" was created in 1968 [90], software components have been the primary foundation for building component-based software/systems (CBS) in component-based software engineering (CBSE) [74] [139] [117] [137]. In recent years, services (e.g. component-based services, web services) and service-oriented architecture [58] have been evolving as a new generation of software components and CBSE, which further shows their importance to the entire software domain.

Software component testing (SCT) [24] [66] is a proven software engineering approach to evaluating, improving and demonstrating component reliability and quality [42] [66] for producing trusted software components [93] [94], which plays a critical role in support of the success of CBSE. A major factor is that inappropriate reuse of untested, defective, unreliable and poor-quality software components may lead to serious software reliability and quality problems. Although component functionality and reusability are always needed, SCT particularly assures component reliability and quality, and thus becomes an integral part of the development lifecycle of software components and CBS. Therefore, the importance of SCT in CBSE cannot be underestimated.

SCT focuses on producing component test cases (CTCs), which is the central part of all SCT tasks. Our literature review (in Chapter 2 and Chapter 3) will show that there are three main categories of commonly-used testing approaches for test design and generation, which are implementation-based testing (IBT), specification-based testing (SBT) and model-based testing (MBT) (especially as reviewed in Section 2.5). Among them, one of our findings is that MBT has more advantages, and can well support component integration and system testing (as reviewed in Section 3.2.7). Furthermore, recent model-driven software development paradigms, such as model driven architecture/development/engineering (MDA/MDD/MDE) [106] [84] [134] together with the standardised Unified Modeling Language (UML) [108] [28] [125], have

enabled MBT to become much more popular, and UML-based testing (UBT) emerges as a new and active mainstream approach to MBT. This thesis adapts MBT/UBT as the primary approach to SCT to accomplish the effective integration of SCT and MBT, and particularly focuses SCT/MBT on UML-based testing of software components and CBS (i.e. UML-based SCT). The main areas of SCT/MBT in this thesis cover UBT and UML-based SCT.

Although MBT has evolved from model-based design/development (MBD), advancing from MBD to MBT poses crucial challenges for the development of useful MBT approaches. The same is true for the development of effective UML-based SCT approaches when advancing SCT with UML from software component design/development (SCD) with UML. Despite much research on the areas of SCT/MBT in the literature, there still exist a number of important challenging problems that remain unresolved and hamper the utilisation of SCT/MBT (as reviewed in Chapter 2 and Chapter 3, especially in Section 3.5). Resolving these problems requires further research to achieve the goal of desirable SCT/MBT effectiveness.

The following outlines the most important research problems that are directly related to the scope of this thesis:

(1) Lacking a unified testing process, where SCT/MBT activities can be integrated properly into the entire software development process (as further reviewed in Section 3.2.5.1, Section 3.2.6.3, Section 3.5 and Section 3.6)

On the one hand, because the use of MBT approaches means a significant paradigmatic change from IBT or other traditional testing approaches, there are some obstacles in technology transfer of MBT into testing organisations, so the overall process of software development and testing must be adapted. On the other hand, because of the aforementioned problem, software models used for test generation are not incorporated appropriately with software artefacts produced from the software development process, or software models are defined merely by and for a specific MBT approach in use. This can cause the use of an MBT approach not to be cost-effective.

This research emphasises the intrinsic connection of MBT to its counterpart MBD. We argue that MBT should be an integral part of, and should be incorporated into, the entire software development process. We also argue that the importance of models constructed for SCT/MBT (i.e. *test models* as termed in this thesis) should be considered equally with models constructed for SCD/MBD (i.e. *design/development models* as termed in this thesis), which could allow model-based component development and testing to collaboratively work together for producing quality components. This suggests that it is very important to investigate a unified SCT/MBT process that can well integrate model-based component development and testing ac-

tivities. It would be also very useful to apply UML modeling in this unified process to enable the utilisation of a consistent model-based approach and specification with UML for all component development and testing activities.

(2) The deficiency in the investigation of how to bridge "*test gaps*" for component testability improvement in MBT/UBT (as further reviewed in Section 3.2.6.3, Section 3.5 and Section 3.6)

There is a misunderstanding that existing development models can be reused directly (without change) as test models for MBT/UBT. This is impractical because ordinary development models by their nature are simply not test-ready or testable. In fact, there exist certain gaps between ordinary development models (which are non-testable) and target test models (which should be testable). A primary reason for the existence of these "*test gaps*" (as termed by this thesis) is that ordinary development models mainly aim for component design and implementation, and accordingly, they may not contain sufficiently adequate testing information to support MBT/UBT. Such "test gaps" are a major obstacle to overcome inadequate model-based component testability, with the result that relevant MBT/UBT activities (e.g. model-based fault detection and diagnosis) can not be carried out properly in the MBT/UBT practice.

Therefore, it is necessary to explore what these "test gaps" are exactly and how to cope with them in the MBT/UBT practice. In particular, it would be useful to investigate a testing technique that can bridge the "test gaps" and improve model-based testability to facilitate relevant MBT/UBT activities.

(3) The deficiency in methodological comprehensiveness in most reported SCT/MBT approaches (as further reviewed in Section 3.5)

In most situations, a SCT/MBT approach reported in the literature has only fewer (usually one or two) individual testing techniques. Consequently, this results in the SCT/MBT approach having limited testing features and capabilities. This is seen in the most reported representative SCT/MBT approaches that have been highly cited by many research papers in the literature (as reviewed in Chapter 2 and Chapter 3). For example, they do not sufficiently cover fault diagnosis and localisation, or lack detailed and operational descriptions about how to generate actual test cases and oracles, etc.

Although it is impractical to have a complete or perfect testing approach, we argue that it would be much more useful to develop a new comprehensive SCT/MBT methodology, which is expected to contain a set of supporting testing techniques and processes, methodological features and testing capabilities to undertake all required SCT/MBT activities. Software testers would wish to have such a comprehensively integrated testing methodology to meet practical testing requirements, which could considerably reduce the costs of method learning and selec-

tion. We argue that methodological comprehensiveness is a key to the success of any SCT/MBT methodology.

(4) Lacking comprehensive validation and evaluation for most reported SCT/MBT approaches (as further reviewed in Section 3.5)

Most SCT/MBT approaches reported in the literature are not evaluated analytically and empirically, and are shown only with some individual testing examples. Lacking comprehensive validation and evaluation for a testing approach does not provide convincing evidence on its practical usefulness, which could make it difficult for software testers to select and apply this testing approach for their testing needs.

We argue that any SCT/MBT approach under investigation should be validated and evaluated comprehensively, for example, by using a series of full case studies. This can not only confirm the testing features and capabilities that are claimed by the SCT/MBT approach, but also demonstrate its advantages over other related testing approaches.

1.2 Research Motivations and Objectives

The primary research motivation of this thesis is to address the four most important challenging research problems as described in Section 1.1. The principal research objective of this thesis is to introduce a novel hybrid UML-based SCT methodology, called *Model-Based Software Component Testing* (MBSCT), to effectively support the most important SCT/MBT activities, especially for test model construction, component test design and generation, component testability improvement, component fault detection and diagnosis (FDD), and component test evaluation, with the ultimate goal to produce trusted quality software components and benefit all component stakeholders (e.g. developers, testers, users) in CBSE. This research focuses on component integration testing (CIT) that bridges component unit testing and component system testing, which particularly supports software component integration (SCI) that is the most common component reuse method employed in CBSE practice.

To address the first challenging research problem, this research introduces a novel unified testing process, called *Model-Based Integrated SCT* process, which aims to integrate UML-based SCT activities with the corresponding UML-based SCD activities as the core phases of the SDLC (software/system development life cycle). This enables the utilisation of a consistent UML-based approach and specification for all SCD and SCT activities under this integrated SCT process. By guiding test model construction and model-based test development iteratively and incrementally, the integrated SCT process is a base methodological component for the entire MBSCT methodology and its framework.

This research addresses the second challenging research problem by identifying "*test gaps*" in MBT/UBT and introducing a novel contract-based testing technique, called *Test by Contract* (TbC) technique, which aims to bridge the identified "*test gaps*" and improve component testability. The TbC technique is another base MBSCT methodological component, and provides a stepwise TbC working process and a contract-based fault detection and diagnosis (CBFDD) method to support test model construction, component test design and generation, component fault detection, diagnosis and localisation.

In addition to the above two base MBSCT methodological components, this research proposes a new scenario-based testing technique, called *Scenario-Based Component Integration Testing* (CIT) technique, which aims to emphasise the importance of identifying and constructing test scenarios and test sequences and to develop scenario-based test models and scenario-based component tests for the CIT purpose. This research also introduces a novel testing strategy, called *Test-Centric Remodeling* (TCR) strategy, which aims to incorporate the TbC technique into model-based testability improvement and the scenario-based CIT technique into test-centric model optimisation in test model construction. Finally, this research introduces a novel mapping-based testing technique, called *Component Test Mapping* (CTM) technique, which provides a stepwise CTM process and aims to guide test mapping and transformation from testing-related component artefacts at different modeling levels towards test derivation of target CTCs.

This research addresses the third challenging research problem with the development of the MBSCT methodology that has the desired methodological comprehensiveness. The MBSCT methodology is developed with the abovementioned *five major methodological components*: the model-based integrated SCT process, the scenario-based CIT technique, the TbC technique, the TCR strategy, and the CTM technique. They jointly support the most important SCT/MBT activities in the *three-phase testing framework* (including test model construction, component test design and generation and component test evaluation), and enable MBSCT to be model-based, process-based, scenario-based, contract-based, FDD-based and mapping-based, which form the *six main MBSCT methodological features*. All these methodological components and features further support the *six core MBSCT testing capabilities*, which are: (1) test model construction, (2) component test design and generation, (3) component fault detection, diagnosis and localisation, (4) adequate test artefact coverage, (5) component testability improvement, (6) adequate component fault coverage and diagnostic solutions.

This research also undertakes two full case studies for the comprehensive validation and evaluation of the MBSCT methodology (including the methodological components, testing framework, methodological features and testing capabilities), in order to address the fourth challenging research problem.

In addition, this research conducts a comprehensive literature review on SCT/MBT and introduces a set of useful new SCT/MBT concepts and definitions to create a solid conceptual foundation for the development of the MBSCT methodology. The MBSCT methodology integrates these new SCT/MBT concepts and definitions into the MBSCT methodological components to consolidate the MBSCT's testing framework, methodological features and testing capabilities.

This research was initially motivated by the previous Software Component Laboratory (SCL) work, which was an Australian Government funded project [40] [96] [97] [98] [88] (Appendix A describes an overview and review of the SCL work). The SCL project proposed an XML-based *component test specification* (CTS) for specifying and representing component test cases (called *CTS test case specifications*), which has several unique characteristics and advantages over traditional test case representations [24] [117] [137] (such as a well-defined and well structured specification format), and is used by the MBSCT methodology. The SCL project also developed an accompanying testing tool, the *test pattern verifier* (TPV) for verifying CTS test case specifications in dynamic testing, which is also used by the MBSCT methodology.

To further the work of the previous SCL project, this thesis intends to address some of the main limitations and remaining issues of the SCL project (as reviewed in Section A.4 in Appendix A), specifically by the investigation of a systematic approach to model-based design and generation of component test cases represented as CTS test case specifications, component integration testing, component testability improvement, and component fault detection and diagnosis.

The significance of this research is to address a set of the most important challenging problems remaining in the SCT/MBT area in general and a number of the main limitations of the previous SCL project in particular. The proposed MBSCT methodology is put forward as our resolution to these problems, which aims to overcome certain obstacles to advance widespread SCT/MBT utilisation and to achieve the desirable SCT/MBT effectiveness. The significance of this research is strongly supported with a set of original contributions achieved by this research, which are described in the next Section 1.3 and revisited in Chapter 10.

1.3 Overview of Original Contributions

This thesis makes substantial and original contributions to the Software Engineering scholarly body of knowledge in the main research areas of software components, software component testing, model-based testing, UML-based testing, contract-based testing, scenario-based testing,

mapping-based testing, and fault detection, diagnosis and localisation. The original contributions comprise two major parts, with respect to the substantial literature review for the solid research foundation and the comprehensive MBSCT methodology developed as the result of this research.

The following presents an overview of the original contributions of this thesis:

1. The original contributions arising from the literature review for the research foundation (in Chapter 2 and Chapter 3)

1.1 In the research areas of software components and software component testing

(1) A new comprehensive taxonomy of software component characteristics (in Section 2.2.2)

(2) A new software component definition (in Section 2.2.3)

(3) A new definition of software component testing (in Section 2.3)

(4) A useful taxonomy of software component testing techniques (in Section 2.5)

(5) A practical taxonomy of component testability improvement approaches (in Section 2.6)

1.2 In the research areas of model-based testing and UML-based testing

(1) A study of model-based tests (in Section 3.2.4)

(2) A new definition of model-based testing (in Section 3.2.5)

(3) A new test model definition (in Section 3.2.6)

(4) A new definition of UML-based testing (in Section 3.3.1)

(5) A core UML subset for SCT (in Section 3.3.2)

(6) A study and review of use case driven testing and scenario-based testing (in Sections 3.3.2 to 3.3.3, and Sections 3.4.2 to 3.4.5)

2. The principal original contributions of the MBSCT methodology (in Chapter 4 to Chapter 9)

The principal original contributions of this research are to introduce a novel hybrid SCT methodology – *Model-Based Software Component Testing* (MBSCT), which is developed to possess five major methodological components, a three-phase testing framework, six main methodological features and six core testing capabilities.

2.1 The five major MBSCT methodological components that have been developed are:

(1) Model-Based Integrated SCT Process (in Chapter 4 and Chapter 5)

(2) Scenario-Based Component Integration Testing Technique (in Chapter 4 and Chapter 5)

(3) Test by Contract (TbC) Technique (in Chapter 4 to Chapter 7)

(4) Testing-Centric Remodeling (TCR) Strategy (in Chapter 4 and Chapter 5)

(5) Component Test Mapping (CTM) Technique (in Chapter 4 and Chapter 8)

2.2 The MBSCT framework has been created as a new model-based testing framework with the following three main phases for undertaking UML-based SCT:

(1) Test Model Construction (in Chapter 4 and Chapter 5)

(2) Component Test Design and Generation (in Chapter 4 to Chapter 8)

(3) Component Test Evaluation (in Chapter 7 and Chapter 9)

2.3 The MBSCT methodology and its framework have six main methodological features.

The MBSCT methodology enables SCT to be model-based, process-based, scenario-based, contract-based, FDD-based, and mapping-based in the SCT practice.

2.4 The MBSCT methodology and its framework have six core testing capabilities.

(1) MBSCT Capability #1: test model construction

(2) MBSCT Capability #2: component test design and generation

(3) MBSCT Capability #3: component fault detection, diagnosis and localisation

(4) MBSCT Capability #4: adequate test artefact coverage

(5) MBSCT Capability #5: component testability improvement

(6) MBSCT Capability #6: adequate component fault coverage and diagnostic solutions

1.4 Thesis Structure and Outline

This thesis is structured into ten chapters and three appendices. After the thesis introduction in this chapter, Chapter 2 and Chapter 3 present the comprehensive literature review and early research results (including the new concepts and definitions as described in Section 1.3). Chapter 4 to Chapter 8 introduce the MBSCT methodology and its framework, and systemically demonstrate how to apply them to UML-based SCL activities with a number of illustrative testing examples. Chapter 9 undertakes further methodology validation and evaluation with two full case studies, followed by the thesis conclusion and the suggestions for future work in Chapter 10.

The outline of chapter and appendix contents in this thesis is described below:

(1) **Chapter 2 Foundation of Software Components and Software Component Testing**

Chapter 2 presents a comprehensive review of important concepts, principles, characteristics and techniques of software components and SCT in the current literature. Based on this,

further research work on software components and SCT is described with a number of further research results (including new concepts and definitions) as part of the original research contributions achieved by this thesis.

(2) **Chapter 3 Model-Based Approaches: Models, Modeling and Testing**

Chapter 3 comprehensively reviews model-based testing, UML-based testing and related work in the current literature. Based on this, further research work on model-based development and testing is described with a number of further research results (including new concepts and definitions) as part of the original research contributions achieved by this thesis.

(3) **Chapter 4 Model-Based Software Component Testing: A Methodology Overview**

Chapter 4 presents an overview of the MBSCT methodology and its framework introduced by this research, which are the principal original contributions achieved by this thesis. The main principles and technical aspects of the five major MBSCT methodological components are described. This chapter also outlines the three-phase testing framework, the six main methodological features and the six core testing capabilities of the MBSCT methodology.

(4) **Chapter 5 Building UML-Based Test Models**

Chapter 5 applies the MBSCT methodology to develop a set of UML-based test models for UML-based SCT in the first phase of the MBSCT framework. This chapter discusses the main tasks and techniques for test model construction with the first four MBSCT methodological components, and demonstrates how to apply them to construct UML-based test models (e.g. use case test model, design object test model) with the illustrative testing examples selected from the first case study, the Car Parking System (CPS).

(5) **Chapter 6 Test by Contract for UML-Based SCT**

Chapter 6 introduces the TbC technique and several important contract-based test concepts (including test contract, Contract for Testability, effectual contract scope, internal/external test contract), and designs a set of six contract-based test criteria (i.e. TbC test contract criteria) for effective testability improvement. This chapter develops a useful stepwise TbC working process, and demonstrates how to put the TbC technique into practice for contract-based testing activities to undertake UML-based SCT, which is illustrated with the selected testing examples from the CPS case study.

(6) **Chapter 7 Test by Contract for Component Fault Detection, Diagnosis and Localisation**

Chapter 7 focuses the TbC technique (especially the advanced phase of the stepwise TbC working process) on component fault detection, diagnosis and localisation. After introducing an

extended fault causality chain and a new notion of *Contract for Diagnosability*, the CBFDD method (including the CBFDD process and guidelines) is developed to guide FDD activities. This chapter analyses important interrelationships between test contracts and fault diagnosis properties in terms of fault propagation scope, fault diagnosis scope and effectual contract scope. Based on this, the CBFDD method is applied to develop fault diagnostic solutions (including direct diagnostic solutions and stepwise diagnostic solutions in two major testing contexts), and to detect, diagnose and locate component faults with the illustrative testing examples selected from the CPS case study.

(7) **Chapter 8 Component Test Design and Generation**

Chapter 8 discusses the main tasks and techniques for component test design and generation with the five MBSCT methodological components in the second phase of the MBSCT methodology. In particular, this chapter introduces the CTM technique, and describes the CTM definition and the stepwise CTM process with the six main CTM steps for component test derivation. The CTM technique is applied to derive target CTS test case specifications, which is illustrated with the selected testing examples from the CPS case study.

(8) **Chapter 9 Methodology Validation and Evaluation**

Chapter 9 reports two full case studies (i.e. the Car Parking System (CPS), and the Automated Teller Machine (ATM) system) undertaken in this research for further validation and evaluation of the MBSCT methodology and its framework. The case studies examine and assess the testing applicability and effectiveness of the six core MBSCT testing capabilities. The result of this methodology validation and evaluation demonstrates and confirms that the six core MBSCT testing capabilities are effective to achieve the required level of component correctness and quality.

(9) **Chapter 10 Conclusions and Future Work**

Chapter 10 concludes this thesis by revisiting the original research contributions with further discussions, and exploring important open issues concerning methodology improvement and research directions for future work.

(10) **Appendix A Software Component Laboratory Project**

Appendix A presents an overview and review of the previous SCL project, which motivated this research to address some of its main limitations and remaining issues.

(11) **Appendix B Case Study: Car Parking System**

Appendix B presents the CPS case study, and provides the background and supplementary information about this case study. The most important aspects of methodology validation

and evaluation with this case study are described in Chapter 9.

(12) **Appendix C Case Study: Automated Teller Machine System**

Appendix C presents the ATM case study, and provides the background and supplementary information about this case study. The most important aspects of methodology validation and evaluation with the ATM case study are described in Chapter 9.

Chapter 2
Foundation of Software Components and Software Component Testing

2.1 Introduction

SCT plays a critical role in support of the success of CBSE and its importance in CBSE cannot be underestimated (as described earlier in Section 1.1). Software components and CBS are the primary subject of software/system under test (SUT) in the scope of this thesis, and SCT (including testing of software components and CBS) is the central focus of this research.

Our study shows a *special CBSE diversity characteristic*: a distinguishing characteristic of component-based software engineering different to the traditional (non component-based) software engineering is that *different stakeholders* (e.g. developers, testers, users, etc.) play *different roles* with *different perspectives* for *different needs*, and work with *different resources* in *different contexts*. This special CBSE diversity characteristic (which is adapted from [166], and will be further discussed in Section 2.2.1.4 and other related sections) influences the approaches for both SCD and SCT in CBSE practice, and poses significant challenges in these important research areas. Accordingly, it is necessary to understand and investigate fundamental aspects of software components and SCT.

Among many aspects, this chapter particularly focuses on the following important issues and concerns of primary interest in software components and SCT:

(1) What is a software component? Why are there different component definitions that contain different component properties in the CBSE domain? (in Section 2.2.1)
(2) What are software component characteristics? What component characteristics support SCT? How do we classify them to develop a proper taxonomy? (in Section 2.2.2)
(3) How do we develop a new component definition to particularly emphasise the importance of software component testing and quality in CBSE? (in Section 2.2.3)
(4) What is software component testing? What are the main characteristics of SCT? What are CTCs and specification? (in Section 2.3)
(5) What are the general SCT process and test levels? (in Section 2.4)
(6) How do we classify SCT techniques to develop a proper taxonomy? (in Section 2.5)
(7) What is software component testability? What are the main approaches to improve testability? How do we classify them to develop a proper taxonomy? (in Section 2.6)

This chapter presents a comprehensive review of important concepts, principles, characteristics and techniques as well as related work of software components and SCT in the literature. Based on this in-depth literature review, we undertake further research work to develop new concepts and definitions, which aims to enrich the relevant knowledge and principles of software components and SCT in the literature. We show our research viewpoints and results to reinforce the importance of component testing and quality in CBSE, which is the central focus of this research. The principal goal of this research in Chapter 2 is to create a solid foundation and proper background in these primary research areas for the development of the new MBSCT methodology by this research.

This chapter is organised into two main parts. The first part is Section 2.2 that reviews a number of different component definitions and characteristics (in Section 2.2.1), and introduces a new comprehensive taxonomy of software component characteristics (in Section 2.2.2). Based on this, we propose a new software component definition (in Section 2.2.3). The second part of this chapter from Section 2.3 to Section 2.6 focuses on SCT. Section 2.3 proposes a new SCT definition, and describes the associated generic testing process and main testing tasks (in Section 2.3.1). We then study and analyse important SCT characteristics (in Section 2.3.2), test cases and specification concepts (in Section 2.3.3), different testing perspectives and needs (in Section 2.3.4), and main SCT limitations (in Section 2.3.5). Section 2.4 describes the main SCT phases and levels in the general SCT process, from individual components to component integration and CBS. Section 2.5 introduces a useful taxonomy of SCT techniques for test design and generation, and correlates them to relevant test levels. Section 2.6 studies and discusses component testability concepts, characteristics, and general strategies to improve component testability. We then develop a practical taxonomy of testing approaches for component testability improvement and show a comparative study from different perspectives. Finally, Section 2.7 presents the summary and discussion of this chapter.

2.2 Software Components

2.2.1 A Review of Software Component Definitions

2.2.1.1 Different Definitions of Software Components

The concept of software components has been active in the computer software community almost for four decades, since it was initially introduced by Dr McIlroy at the 1968 NATO Software Engineering Conference [90]. However, the question of “what is a software component?” is not simple with a definitive answer. There are numerous definitions about software compo-

nents in the literature [39] [38] [74] [44] [139] [94] [155] [66] [127] [87]. Table 2.1 illustrates some of the important component definitions given by the well-known researchers/organisations in the literature.

2.2.1.2 Review and Analysis

It is necessary to study and review existing component definitions, and identify and evaluate the essence of common software components, in order to answer the above question appropriately. To effectively analyse and evaluate existing component definitions, we extract and summarise the key software component characteristics that are directly/indirectly involved in the respective definitions, as shown in Table 2.1 (Section 2.2.2 will further discuss software component characteristics in detail).

Table 2.1 Review of Software Component Definitions

Definition Reference Source	Definition Description	Component Characteristics
Definition by Booch [27]	A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.	Reusability, modularity (cohesive, coupling), encapsulation (abstraction)
Definition in OMG UML v1.5 [107]	A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.	Modularity, deployable, replaceable, encapsulation, interfaces
Definition by Heineman & Councill [74]	A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.	Component model, independent deployment and composition, composition standard
Definition by Meyer [94]	A component is a software element (modular unit) satisfying the following conditions: 1. It can be used by other software elements, its "clients." 2. It possesses an official usage description, which is sufficient for a client author to use it. 3. It is not tied to any fixed set of clients.	Modularity, usability/reusability, usage interfaces, independent use
Definition by Szyperski [139]	A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.	Composition, contract-based interfaces, context-dependencies, independent deployment, third-party composition

There are some common component characteristics in the existing component definitions, such as modularity, reusability, interface, etc. Furthermore, we can see that different definitions take different viewpoints and focus on different aspects of software components. Accordingly, each definition holds some key component properties from its particular perspective.

(1) Booch's [27] definition seems relatively simple.

This definition has three basic component attributes of reusability, modularity and encapsulation (abstraction).

(2) The component definition in OMG UML v1.5 [107] has a similar approach.

This definition includes a few more component properties of modularity, encapsulation, interfaces, deployable and replaceable attributes.

(3) Heineman & Councill [74] base their definition on some aspects of high-level component standards.

This definition requires software components to have conformance to component models and composition standards, in addition to some necessary component properties of independent deployment and composition.

(4) Meyer [94] takes a broad view of components with certain characteristic-based conditions.

This definition views a broad range of (small/large-grain) modular units as components, if they fulfil the three conditions on important component characteristics of usability/reusability (condition #1), usage description or interface specification (condition #2), and independently usable by any client or independency (condition #3).

(5) Szyperski [139] gives a more rigorous and precise component definition.

This definition emphasises the key component characteristics, such as composition, contract-based interfaces, context-dependencies, independent deployment, and third-party composition perspective. But the definition does not mention any component models and standards. Among many other definitions in the literature, Szyperski's definition is well-known and widely-accepted in the CBSE community.

2.2.1.3 Component-Related Stakeholders

When studying software components and their definitions, we need to consider another important factor associated with component-related stakeholders. We can classify component stakeholders into three main categories in the CBSE domain, as shown in Table 2.2.

(1) Component production stakeholder: developer, tester, quality engineer, project manager.

These stakeholders are the major producers or builders of software components in CBSE.

(2) Component trade stakeholder: trader/vendor.

Note that a tester may also be part of the component trade stakeholder group because it

may be necessary to undertake final testing just before a component sale to customers.

The above two types of component stakeholders (1) and (2) are jointly called component providers/suppliers according to the primary role they serve for component users.

(3) Component consumer stakeholder: user/customer.

The component user is the final stakeholder who finally purchases, uses/reuses and operates software components in the CBSE domain.

Table 2.2 Component-Related Stakeholders

Stakeholder			Description (Role/Perspective/Need)	Resource	Context	Relation -ship
Provider	Producer	Developer	Analyse, design and implement components.	development information	development environment	production member
		Tester	Test, verify and validate components.	testing information	testing environment	production member, or trade member
		Quality Engineer	Standardise, measure and evaluate component quality; certify and ship components.	quality information	quality environment	
		Project Manager	Plan, manage and coordinate components project.	management information	management information	production member
	Trader /Vendor		Manage component repository; trade and sell components.	trade information	trade environment	trade member
User/Customer			Select, reuse, integrate and deploy components; build, use and operate CBS.	use/reuse /deployment /application information	use/reuse /deployment /application environment	use/reuse member

2.2.1.4 Special CBSE Diversity Characteristic

Based on the above study and review (in Section 2.2.1.1 to Section 2.2.1.3), we can conclude a special CBSE diversity characteristic as defined earlier in Section 2.1. As shown in Table 2.2, it is clear that different stakeholders hold different perspectives towards software components. One may see and get different component definitions from different component stakeholders who have different needs or requirements for good components. Such requirements are closely related to what characteristics components should have, in order to fulfil all stakeholders' needs, especially for the component users who finally purchase, use and operate software components.

Accordingly, this special CBSE diversity characteristic is a primary reason why there are different component definitions that contain different component properties in the CBSE domain. Another important reason for this is that the concept of software components itself has actively evolved gradually from its early stages towards more maturity nowadays, along with many different concepts and techniques of SCD and CBS design/development that have emerged for building software components and CBS.

Currently in the CBSE domain, there is no single formal component definition. Furthermore, there is no standard that specifies what is a "good" software component, what is the standard for component models, what is the standard of component infrastructure and framework, and so on. Because of the lack of standardisation, software engineers can only take advantage of some key component characteristics to a limited extent in CBSE practice [66]. Section 2.2.2 will discuss software component characteristics in more detail.

2.2.2 A New Taxonomy of Software Component Characteristics

Generally speaking, software components should have a number of characteristics and properties that can denote and reflect component functionality, quality and performance relevant to all component-related stakeholders, and especially deliverable to component users [137]. So what are "good component characteristics"? Accordingly, it is necessary to study and review component characteristic aspects in the literature [39] [74] [154] [44] [139] [155] [66] [87] in order to identify "good component characteristics" to establish a component quality metric for measurement of "good software components". This is an important aim of this research to further the knowledge of component concepts and principles in the literature (as described earlier in Section 2.1).

In this section, we introduce a new comprehensive taxonomy of component characteristic properties (as shown in Table 2.3, Table 2.4 and Figure 2.1 below). A major goal of this new taxonomy is to establish a proper component quality metric for the determinant of what a "good software component" is. Another important goal is to apply this new taxonomy to guide software component testers to focus on the crucial component characteristics during testing. Note that the list of component characteristics in the taxonomy is intended to be neither completely inclusive nor exclusive to other classifications in the literature. The important purpose here is to provide a solid foundation for systemically studying component characteristics, developing a new component definition for effective component testing.

The following subsections discuss in detail component characteristic classification (in Section 2.2.2.1), interrelationship (in Section 2.2.2.2), and new component properties (in Section 2.2.2.3) (which we have identified and added to the taxonomy, as shown as the asterisked items in Table 2.3, Table 2.4 and Figure 2.1 below).

2.2.2.1 A Classification of Software Component Characteristics

Our taxonomy includes twenty-two (22) software component characteristics. We classify these component characteristics into four (4) main categories, as described in Table 2.3 (Taxonomy Part 1). The first category describes essential functional component properties, and the other

three categories describe non-functional component attributes.

Table 2.3 Taxonomy of Software Component Characteristics (Taxonomy Part 1)

Level	Characteristic	Description
Implicit/essential	Functionality	Well-defined, dedicated capability that provides functions and services to fulfil specified requirements. Functionality features usefulness and values that are most important to all stakeholders.
	Executability	The capability of being executed to perform required functions in the specified context. Executability is a prerequisite of functionality and other related properties to be achievable and deliverable in execution.
	Usability	The ease of use of component deliverables as expected and satisfied. Usability requires the capability of being learned, understood and operated, and the efficacy of use from the user perspective.
Basic	Identity	The unique representation of a component so that a particular component can be differentiated from other peers, and can be distinctively identifiable in the lifecycle contexts of development, testing, reuse, deployment, operation, maintenance and repository. Identity can be represented with a well-defined naming scheme for the distinguishing identification.
	Modularity	The extent of being composed of individually distinct units (called modules). Good modularity requires high cohesion and low coupling, which is a key requirement of a module to be one or part of a component.
	Encapsulation	Enclosing related representation and implementation in one unit of organisation. Encapsulation hides internal working information (e.g. implementation and data) to be externally invisible and inaccessible, except external interfaces. Typical units of encapsulation are objects, classes, modules and packages.
	Interface	Abstraction of component services with externally visible operational specifications (e.g. publicly accessible operations' signatures, but not their implementation details). Interfaces are access points to functional services by external clients, and provide a common interconnection between two or more components for interactions and communications.
	Independence	Separation of responsibilities from operational environments for integration and deployment; being delivered as independent parts so that they can be replaced under certain conditions and constraints.
	Reusability	The capability of ease of reuse by different clients in different application environments. Component reuse can be as a whole or in part (ideally without modification). Typical reusable component elements include functions, interfaces, specifications, source code, executables, test cases, user manuals, etc., but not just executable programs only.
	Portability*	The capability of being platform-independently ported and executed from one computer system environment to another (ideally without modification).
	Documentation	Specifying and documenting software elements, including software documents for specifications, interfaces, reuse, deployment, user manuals, etc. Component elements should be documented for effective use/reuse.
Intermediate	Customizability	The capability of modifying software artefacts to meet individual customer needs and/or particular operating environment requirements. Customization selects, tailors and configures component functions, interfaces and other related elements, and then packages customized component elements for a new delivery. Customizable components hold enhanced reuse and deployment capabilities.

	Deployability	The capability of software distribution to put into use and operation from development and/or third-party sites into the targeted operating environment. Deployment customises (if applicable), packages, installs and activates executable component instances to be ready for execution and use in the runtime environment. It is the final stage of realising component reuse in the new target environment.
	Interoperability	The capability of supporting intercommunications and data/message exchanges between peer components across different processes on the local computer system or over the network system. Interoperable components jointly fulfil communications and collaborations required in integration contexts.
	Composition	Composing parts into the whole (ideally without modification) to construct complex components and systems. Composition reuses and assembles composite components or parts for component integration. Composition relationship is transitive.
	Integrability*	The capability of being integrated to develop compound components and systems in the operating environment. The integration process includes customisation, composition, configuration and other activities to combine and unite all software and/or hardware components into an overall executable system.
	Substitutability*	The capability of replacing a component with another in the same or different contexts under certain conditions. The permitted substitution requires that the substituting component fulfils the equivalent features of the substituted component (e.g. functions, interfaces) and usually has certain improved effects (e.g. better performance or reliability).
Advanced	Testability*	The extent of the ease of being tested for conformance to certain testing requirements. Testable components facilitate the establishment of test criteria and performance of tests to determine whether those criteria have been met. Strong testability indicates the ease of observing and controlling test inputs and outputs to enhance testing effectiveness and efficiency.
	Reliability*	The capability of a component/system that can fulfil the required functions and maintain the level of performance consistently and satisfactorily under stated conditions. Reliability requires correctness and robustness, and denotes the probability of failure-free operation in a specified environment for a specified period of time.
	Integrity* (security, safety)	The condition/state/quality of being unimpaired, authentic or perfect with the protection against software damage and danger, so that a component/system is able to control and protect its programs and associated data against unauthorised access and malicious attack (e.g. modification, deletion), and to prevent inadvertent and hazardous operations, and accidental failure, injury and life risk. Integrity holds two key associated aspects of security and safety, and enforces protection mechanisms and procedures (e.g. authentication, access control, encryption) to assure software security and safety as well as deliverable correctness and reliability.
	Selectability*	The ease with which a particular component can be evaluated and acquired from a candidate set of potential components for reuse and construction of other components and systems. Selection measures include functional and non-functional factors with many unique component properties to ensure functional and quality components are selected properly.
	Standardised /Standardisation /Standards	Standardising components and associated activities to conform to uniform standards and models for development, testing, quality assurance, reuse, deployment, project management, etc. Component standardisation establishes certain mandatory requirements and measurements that enhance component characteristics for building standardised components.

The four main categories of software component characteristics are described as follows:

(1) Implicit/essential component characteristics (3 characteristics)

These component characteristics are the most essential functional properties that all software components should implicitly possess. In principle, software components share some essential characteristics with good computer software. In particular, software components should be executable, usable and deliver the required functionality.

Note that, although this category of component characteristics is not classified separately and discussed explicitly in the present literature [74] [44] [139] [66], we particularly highlight their significance with our taxonomy. For important SCT objectives (SCT will be discussed in Section 2.3 and onwards), SCT focuses on testing software *executable* programs for functional testing of software components and CBS, which is a major focus of this research.

Besides the essential functional properties, we have identified nineteen (19) non-functional component characteristics that support functional delivery with good quality and performance. These component characteristics can be further classified into three main categories based on the level of componentisation quality: basic, intermediate, advanced.

(2) Basic component characteristics (8 characteristics)

The level of basic component characteristics represents necessary attributes for software components. They serve as a basis for higher-level component characteristics.

(3) Intermediate component characteristics (6 characteristics)

The intermediate level of component characteristics reflects certain desired quality features of software components above the basic level.

(4) Advanced component characteristics (5 characteristics)

Advanced component characteristics stand at the top level of the proposed taxonomy and are required particularly for high quality software components.

Among these component properties in our taxonomy, there exist several *unique* component characteristics that can demonstrate that software components are explicitly different from ordinary software modules, units or other pieces of software systems. These distinguishing component characteristics typically feature in high quality software components, and mainly include reusability, customizability, interoperability, deployability, composition, integrability, substitutability, selectability, etc. Some of these are new component characteristics, which are added to the taxonomy (as shown as the asterisked items in Table 2.3) and will be further discussed in Section 2.2.2.3.

2.2.2.2 Interrelationship among Software Component Characteristics

In this taxonomy, we have given a concise description for each component characteristic, as

shown in Table 2.3. Furthermore, we correlate each component characteristic to other relevant characteristics (if applicable), as shown in Table 2.4 (Taxonomy Part 2). The characteristic interrelationship indicates that a particular component characteristic is either working "*for*" or supported "*by*" some related component characteristics in terms of component characteristic correlations. For example, the characteristic "deployability" is working *for* the feature "reusability" and supported *by* the attributes "independence", "customization", "composition", etc.

Furthermore, we also correlate a particular component characteristic to one or more component-related stakeholders (if applicable) in the taxonomy as shown in Table 2.4. The component user has the major role of stakeholder because software components are built for use by either internal users (e.g. corporate business departments) or external users (e.g. third-party customers). Thus, the characteristic correlation to the component user is especially important, which is illustrated with "*esp. user*" in addition to the ordinary correlation "*All*" for all component-related stakeholders as shown in Table 2.4.

In addition to the textual descriptions, we also give a diagrammatic representation of the taxonomy as shown in Figure 2.1 (Taxonomy Part 3), to aid visualisation of interrelations and their related levels of component characteristics in the taxonomy. A series of right block arrows indicates that component properties proceed from a low level to a high level toward the highest level of component standardisation. Conversely, as illustrated with left block arrows, component standardisation implies that a number of supporting component attributes are included.

Among many component properties, *reusability* is one of most important component characteristics. This component feature is supported by most basic and intermediate component characteristics for effective component reuse. It should be noticed that the "standardised" feature possesses a special mutual interrelationship with other component properties. Good component properties can be united together to support the establishment of a characteristic foundation of component standards; on the other hand, component standards can standardise component models and CBSE processes, and promote good component features across all component stakeholders, so as to produce high-quality software components. From Figure 2.1, complete standardisation is seen to be the ultimate goal for achieving high quality software components in CBSE.

Note that our taxonomy shows the basic common interrelationships among component characteristics in general cases. However, when identifying some interrelationship between two component properties, it is indeed quite difficult to absolutely say that a component property is just only working *for* (or equivalently, supported *by*) another component property, but definitely not vice versa; or they are mutually independent or exclusive without any connection at all. In some case, two component properties (e.g. composition and integration) may work together and/or mutually support each other in one way or another. Investigating the precise interrelationships and *orthogonality* of all component characteristics is useful and important. However,

such a study is beyond the current scope of this research.

Table 2.4 Taxonomy of Software Component Characteristics (Taxonomy Part 2)

Level	Characteristic	Related Characteristics	Stakeholder
Implicit /essential	Functionality	For: all By: executability, usability	All, esp. user
	Executability	For: all, esp. functionality, usability	All, esp. user
	Usability	For: all, esp. functionality By: interface, documentation	All, esp. user
Basic	Identity	For: reusability, selectability	All
	Modularity	For: encapsulation, reusability, testability	All
	Encapsulation	For: reusability, testability By: modularity, interface	All
	Interface	For: reusability, substitutability, testability, selectability By: encapsulation	All, esp. user
	Independence	For: reusability, deployability, substitutability, testability By: portability	All, esp. user
	Reusability	For: selectability By: most basic & intermediate properties, esp. independence, portability, customizability, deployability, interoperability, composition, integrability, substitutability, testability, reliability, integrity	All, esp. user
	Portability*	For: executability, independence, reusability	All, esp. user
	Documentation	For: all	All, esp. user
Intermediate	Customizability	For: reusability, deployability, integrability	All, esp. user
	Deployability	For: reusability, substitutability By: independence, customizability, integrability	All, esp. user
	Interoperability	For: reusability, integrability	All, esp. user
	Composition	For: reusability, integrability By: customizability	All, esp. user
	Integrability*	For: reusability, deployability By: composition, interoperability	All, esp. user
	Substitutability*	For: reusability, deployability, selectability	All, esp. user
Advanced	Testability*	For: reusability, reliability, selectability By: modularity, encapsulation, interface	All, esp. user
	Reliability*	For: reusability, selectability By: testability	All, esp. user
	Integrity* (security, safety)	For: all By: testability, reliability	All, esp. user
	Selectability*	By: all, esp. functionality, reusability, testability, reliability	All, esp. user
	Standardised /Standardisation /Standards	For: all By: all	All

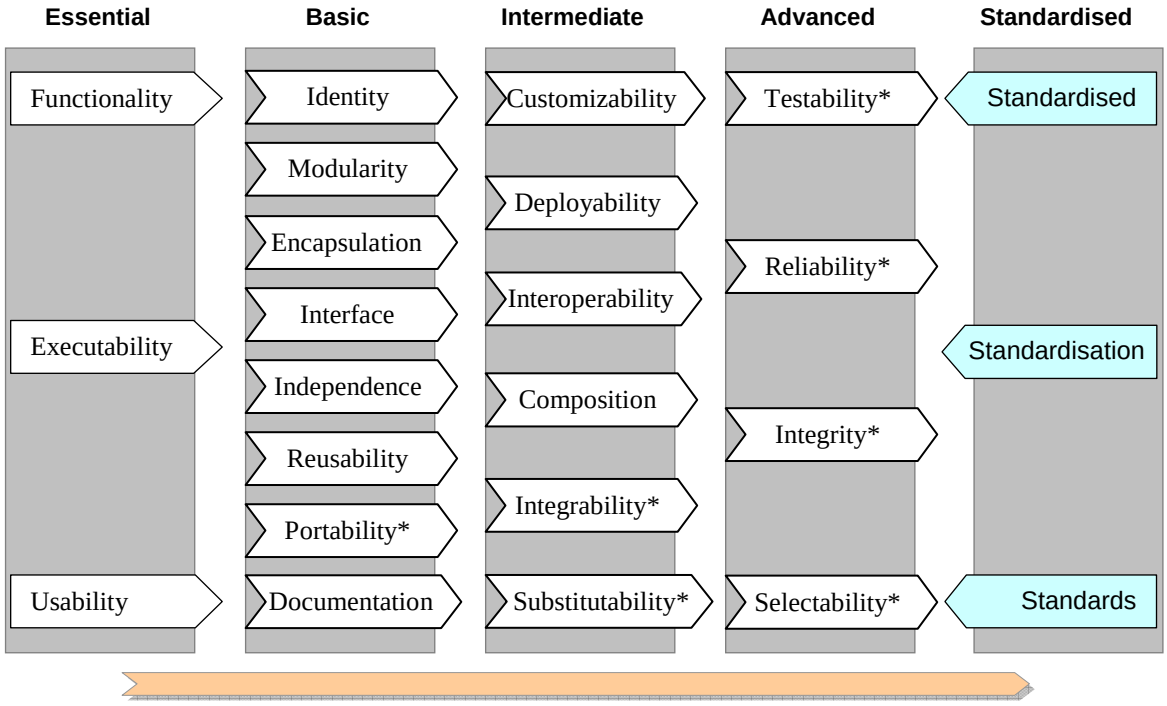


Figure 2.1 Taxonomy of Software Component Characteristics (Taxonomy Part 3)

2.2.2.3 New Software Component Characteristics

In this section, we study a number of new component characteristics in the taxonomy (marked with the star symbol "*" in Table 2.3, Table 2.4 and Figure 2.1), which are not described in the current literature [74] [44] [139] [66]. In the taxonomy, the seven (7) new component characteristics are identified and added to the three main categories respectively as follows:

(1) In the category of basic component characteristics: portability

Software components need to be reused in various computer environments across different platform systems. Component *portability* is necessary for effective component reuse in these diverse reuse contexts.

(2) In the category of intermediate component characteristics: integrability and substitutability

Software components are often used in new integration contexts for building new CBS. No component integration implies no component reuse. Good component *integrability* can enable effective component integration, reuse and deployment. This research examines and evaluates component integrability in the component integration context in conjunction with component integration testing, which is a central focus of our SCT methodology (to be described later from Chapter 4 and onwards).

Substitutability facilitates component replacement to meet the component users' varied needs, such as component reuse, selection and maintenance. In general, it is quite common and reasonable for the component users to substitute an existing software component with a new software component that has equivalent functions and better quality in an existing or new reuse context. A particular component user may do component replacement if the new equivalent software component is more consistent with the integration context (e.g. has the same or compatible computer language implementation and/or runtime environment).

(3) In the category of advanced component characteristics: testability, reliability, integrity and selectability

These advanced component properties specify high quality features of good software components in addition to all other component attributes, which is crucial to the success of CBSE. A major purpose is to minimise and prevent poor selection and reuse of non-testable, unreliable and insecure/unsafe components. Good *testability* can enable relevant component properties to improve the ease of being assessable and predictable, and particularly aid in examining and evaluating component functionality and *reliability*. Testable and reliable components hold high interest for component selection to build CBS effectively, and support component *selectability*. The same characteristic of component *selectability* is also important for component *integrity* particularly in high safety and security CBS.

These new component characteristics (especially the advanced characteristics) introduced with our taxonomy aim to support the delivery of high software quality for component reuse, integration and deployment, and conform to the expected component requirements, specifications and performance. A major focus of this research is on evaluating and improving component testability and reliability to produce quality software components, so that they can be selected, reused and integrated effectively and efficiently.

According to Meyer [93] [94], *trusted components* are the combination of reuse with a special attention to the quality of the components being reused. Our taxonomy developed with the new component characteristics aims to establish a measurement base of guaranteed-quality components, in order to build and provide the trusted components with necessary quality-specific characteristics for the software industry.

2.2.3 A New Software Component Definition

A simple component definition is that a software component is a reusable software unit for building other components and software systems. However, to provide effective reuse and construction capabilities, software components need to possess certain good component characteristics. Moreover, software components also need to have some additional high-level properties

that can effectively support and enhance reliability and quality. Therefore, based on our study on the component definitions (in Section 2.2.1), characteristics and taxonomy (in Section 2.2.2), we can propose a new software component definition as follows:

Definition 2–1. A *software component* is a functional, reusable, testable and reliable software unit with specified interfaces and operating contexts for software composition, interoperation, integration and deployment.

The new component definition covers common component characteristics in the existing component definitions (as reviewed in Section 2.2.1). Moreover, the new component definition has certain important implications and aspects that need to be discussed. The discussion is conducted particularly in conjunction with the new taxonomy of component characteristics developed in Section 2.2.2.

(1) This definition first emphasises *functionality*, which is the most essential component property as shown in the taxonomy. No functionality means no interest in use/reuse to all stakeholders. A particular software component is selected for possible reuse firstly because the component user is interested in its functions before considering any other software aspect.

(2) This definition emphasises *reusability* as the second most important component property. The potential benefit of software reuse is one of the primary reasons for developing and using software components in CBSE.

(3) This definition includes some important and enhanced component characteristics that can support effective software reuse to build complex software components and CBS. These component properties consist of *composition*, *interoperation*, *integration* and *deployment*, which cover most of the important software activities in CBSE. These characteristics are unique to software components and not found in traditional software, as discussed in the taxonomy.

(4) *Specified interfaces* provide useful mechanisms for component reuse, interoperation and integration. This feature allows a software component to be reused as a whole, not partially, with no need to access the internal construction details encapsulated in the component unit. In other words, software components are reused merely through their specified interfaces. This feature supports software component testing, especially for black-box and functional testing based on the component interface specifications.

(5) *Operating contexts* specify component environments where a particular component is reused, integrated and deployed as well as operated. These component contexts may be any kind of virtual, simulated or runtime environments in component-based systems. This feature also supports software component testing, because software components are tested in a similar integration and/or operating contexts where they are used/reused.

(6) *Testability* and *reliability* are the advanced component characteristics as shown in the new taxonomy. In practice, the component users desire to know how well software components are developed and conform to the required functionality and quality. These value-added component properties particularly emphasise important software quality features, and address a verifiable and measurable extent of software quality that would be satisfactory to the component users. Accordingly, these important component properties can effectively assist the users in selection and reuse of testable and reliable components.

The proposed component definition extends the definition coverage scope and contains the most important component characteristics that are conceptually supported by most component properties at the basic and intermediate levels in the new taxonomy (in Section 2.2.2). Based on our literature review, this new component definition appears to be the most comprehensive available in terms of the range of important component characteristics covered, and has a significant advantage over most existing component definitions (as reviewed earlier in Section 2.2.1). Based on this new component definition, our MBSCT methodology is developed to improve component testability and quality, which is one of the major objectives of this research.

Note that the new component definition applies to the different types of component software, which covers common functions/procedures, abstract data types, object-oriented classes, individual components, integrated or complex components, and even component-based systems. This research addresses testing of all these types of component software.

2.3 Software Component Testing

Among many other factors, the success of CBSE relies on not only functional and reusable software components, but also reliable and high-quality software components for building CBS effectively and efficiently. SCT has been shown to be a proven approach to examining, improving and demonstrating the reliability and quality of software components and CBS in practice [16] [24] [100] [66]. SCT is the central focus of this research. After reviewing software components in Section 2.2, we move on to study the foundation aspects of SCT from this section onwards.

2.3.1 Definition of Software Component Testing

2.3.1.1 Existing SCT Definitions

There are many publications and much research effort with regard to SCT, but there is no single formal SCT definition that has been widely accepted and used in the SCT domain. For the question of "what is software component testing?", most existing SCT definitions were based on traditional software testing at the unit level, i.e. SCT is basically treated as traditional unit testing of software modules. For example,

(1) According to Gao et al. [66], software component testing refers to testing activities that uncover software errors and validate the quality of software components at the unit level. In traditional software testing, component testing usually refers to unit testing activities that uncover software errors in software modules.

(2) According to Sommerville [136], component (or unit) testing is that individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components.

(3) The IEEE Standard Glossary [77] gives a definition of component testing as follows: the testing of individual software components or groups of related components.

From a quick analysis of the existing SCT definitions, we can observe some implications and limitations as follows:

(a) The SCT definition by Gao et al. [66] involves two traditional testing aspects of verification and validation clearly at the unit level.

(b) The SCT definition by Sommerville [136] mainly relates to testing of a single component.

(c) The SCT definition by the IEEE Standard Glossary [77] is still very simple, while it considers an aspect of an extended SCT scope over a single component.

In CBSE, SCT and unit testing have some similarities, but they are not exactly same. Among many other factors, the differences between them mainly come from the concept of software components, which evolves from a simple software unit to an entire CBS (as described in Section 2.2.3). This means that a good SCT definition must cover all important testing aspects of different types of component software.

2.3.1.2 A New Definition of Software Component Testing

After a brief review of existing SCT definitions (in Section 2.3.1.1), we propose a new SCT definition in this research as follows: (which is adapted from [174])

> **Definition 2–2.** *Software component testing* denotes a set of software testing activities that analyse component artefacts, design and generate component tests, detect and uncover component faults, and evaluate component reliability and quality of software components and systems under test.

In common with all software testing, the underlying implication of this definition indicates a generic testing process with six major testing phases to carry out key testing tasks as follows:

(1) Component analysis and test planning: to produce test plans and management documents based on component analysis. The test plans describe: (a) testing objectives and requirements; (b) strategies and approaches; (c) resources, costs and schedules, etc.

(2) Component test design and generation: to develop component test specifications that describe test inputs, execution conditions, and expected outputs/results for each CTC.

(3) Component test execution: to execute and operate components/systems with test cases in target testing environments.

(4) Component testing observation and examination: to observe actual test results, analyse the observed test results against CTCs (especially against the relevant expected test results), and examine component functions and behaviours.

(5) Component fault detection: to detect and uncover possible component faults based on component testing observation and examination.

(6) Component testing evaluation: to assess and determine component reliability and quality against component specifications and testing objectives.

Note that the new SCT definition goes beyond the traditional scope of SCT at the individual unit or single component level. In this research, we use the term *SCT* to generally describe the core testing activities for a single component under test (CUT), individual components, integrated components and CBS. Our MBSCT methodology integrates this new SCT definition and particularly focuses on the testing of integrated components and CBS.

2.3.2 Main Characteristics of Software Component Testing

In principle, SCT shares common characteristics of general software testing as described in Table 2.5. This table contains the four main testing characteristics, which are adapted from the *IEEE Software Engineering Body of Knowledge* [138].

Table 2.5 Software Component Testing Characteristics

Characteristic	Description
Dynamic	This characteristic pertains to dynamic testing over static testing. *Dynamic testing* detects software faults with the aid of computer systems, and requires the actual execution of the SUT's program with test inputs to evaluate functions and behaviours in the real runtime or simulated target environment. This requires test cases being executable. By contrast, *static testing* is performed without running the operational program of the SUT to uncover potential inconsistencies and incompleteness in requirements and specifications, and is typically used in early development stages prior to existence of the SUT's executable code. Testing should be ultimately dynamic on the SUT's implementation to meet the user's real needs, because the user actually makes use of the SUT's program for real-life business requests, instead of specification documents.
Finite	This characteristic pertains to finite test space over exhaustive testing. The entire test space could be theoretically infinite with too many test cases due to all possible combinations of program data and condition paths, making *exhaustive testing* impossible and infeasible even for trivial (small or simple) programs. The *finite test space* allows a limited number of test cases to be executed for actual testing within limited testing time and resources. Testing is non-exhaustive and based on finite test cases.
Selected	This characteristic pertains to properly selected test cases from a vast or infinite test space. Testing needs good test techniques that can guide suitably identifying and selecting finite test cases based on certain *test criteria of test selection and coverage* for desired testing effectiveness and efficiency. Well-selected test cases imply cost-effective testing in order to reveal more faults with the selected test cases.
Expected	This characteristic pertains to expected test results with test oracles. Testing needs to determine test pass or fail for each test execution to evaluate expected software reliability and quality. This requires a special test mechanism called a *test oracle*, which is a test generator to produce the expected test results for a specified test input, and a test comparator to compare and check the actual test results against the expected test results. The observed function or behaviour can be examined against requirements (*validation*) or specifications (*verification*). Thus, the expected testing is determined (validated or verified).

The new MBSCT methodology introduced by this research focuses on designing and generating *finite* component test cases and *expected* test results for *dynamic* testing of software components and systems, and applying selected test cases to detect and diagnose component faults.

2.3.3 Component Test Cases and Specification

From the SCT definition and characteristics as described in Sections 2.3.1 and 2.3.2, we can see that SCT activities are driven by component test cases that form the central part of all SCT tasks. This section describes important terms and concepts of component test cases and specification, which is adapted from [165] [166].

Conceptually, a *component test case* (CTC) specifies the CUT's initial state, test environment, test inputs, test execution conditions, expected test outputs/results designed for a particular test objective, such as causing failures, detecting faults, or examining functions. There are three important parts in the CTC concept:

(1) The *test input* specifies test data input to the CUT to discover possible faults or verify specific outcomes as expected.

(2) The *expected test result* is a description of what expected output will be produced by execution of the CUT with the associated test input.

(3) The *test execution* is running a test on the CUT, where the CUT gets the test inputs specified by the CTC, and the actual test outputs are observed and evaluated against the expected test results specified by the same CTC. The testing can be evaluated with a test oracle that is the test generation and comparison mechanism (as introduced in Table 2.5). Test oracles are developed mainly based on software requirements/specifications, and/or testing knowledge and experience of the tester. Test oracles may be manual, automated or partially automated.

A test case is a specification of one scenario to test the CUT. A *test set* (sometimes referred to as a *test suite*) is a collection of test cases that are typically related and organised into a sequence of test cases for a specific testing purpose for the CUT. The related test cases or test sets constitute the core part of a *test specification*, which is a software specification that describes, specifies and represents all testing-related artefacts associated with all testing activities. Several types of test documentation are derived from the test specification, including test plans, test requirements, test design, test environments, test cases, test execution, test evaluation and analysis reports (e.g. faults, errors and repair descriptions), etc. The test specification of the CUT should specify all test cases or test sets of the CUT. Because CTCs are the central part of all SCT tasks and a clear focus of this research, we will often refer to test specifications or test cases or test sets, and these terms are used in appropriate contexts in this research.

2.3.4 Different Perspectives and Needs in Component Testing

All testing activities are mainly conducted by testers or testing tools operated by testers. Because of the special CBSE diversity characteristic (as described in Sections 2.1 and 2.2.1.4), *different roles* of software component testers work with *different resources* in *different contexts*, and thus take *different approaches* towards SCT activities. This indicates that SCT is more challenging than ordinary software testing. Accordingly, it is necessary to understand the *different perspectives* and *needs* of the various stakeholders towards component testing, which are summarised in Table 2.6 (which is adapted from [165]).

This research mainly focuses on functional testing approaches to support the common needs of both types of component testers, as indicated with Table 2.6. Moreover, the user-side tester could employ specification-based functional testing approaches for SCT, in the same way that the production-side tester does, if the user-side tester could access component specifications (e.g., which may be packaged and provided with the component product on request).

Table 2.6 Different Perspectives and Needs Towards Component Testing

Different Roles	Different Resources	Different Contexts	Different Approaches
Testers on the component production side	Have unrestricted access to: (1) full component specifications (e.g. software models); (2) implementation (e.g. programs or source code)	Work in the development environment with strong technical support: hardware, software and technical staff	Can use all possible testing approaches at different testing levels: (1) structural testing approaches (e.g. verification techniques); (2) functional testing approaches (e.g. verification and validation techniques)
Testers on the component user side	(1) Have restricted access to limited informal specifications only on functions and interfaces; (2) Have no access to analysis and design specifications, implementation (programs or source code)	Work in the deployment and/or application environment, with limited or no technical support	Have to use functional testing approaches (e.g. verification and validation techniques), based on only available limited component information in the user's target environment

2.3.5 Limitations of Software Component Testing

SCT aims to examine and evaluate component correctness and quality with numerous advantages. However, SCT also shares certain technical and non-technical limitations with general software testing. It is necessary to emphasise some of the main testing limitations as follows:

(1) Complete or exhaustive testing is infeasible [16] [24]

Despite testing costs being extremely high (see (3) below), complete or exhaustive testing is practically unattainable, because of the testing characteristics as described in Table 2.5 (e.g. Finite, Selected). In other words, testers can only achieve as much test coverage as possible under certain constraints in practice (e.g. time and cost).

(2) Testing is not decidable

As Dijkstra states, "Program testing can be used to show the presence of bugs, but never to show their absence!" [45] In other words, testing cannot show the absence of defects, and it can only show that software defects are present [70]. This implies that some software defects may remain undetected. On the other hand, because complete/exhaustive testing is unachievable in practice (see above (1)), some software artefacts may remain untested, where some software defects hide out. Accordingly, testers cannot guarantee that any tested software is completely 100% correct and perfect.

(3) Testing is labour-intensive and expensive

Testing is time-consuming and a major part of the entire SDLC. Testing costs a large amount of human resources, management effort and budget, compared to other development activities and phases. In particular, studies show that testing can consume more than fifty percent of the total software development costs [16] [70] [21]. As software complexity and criticality grow continuously, testing becomes more and more expensive and difficult.

Due to the testing limitations described above, the next best testing that testers can attain is to carry out "*adequate testing*" to fulfil the practicably achievable testing objectives and requirements. The MBSCT methodology developed in this research takes this approach to undertake SCT activities.

2.4 Software Component Testing Process and Levels

In the same manner as the development process for software components and systems, the general SCT process includes a number of test levels at different testing phases, which are described in Table 2.7 (which is adapted from [165] [167]. The lower three test levels focus on testing of a single CUT, which covers from component operation testing, class unit testing up to inter-class integration testing for the CUT. The upper two advanced test levels go beyond the scope of a single CUT and focus on testing of inter-component integration and component-based systems.

Table 2.7 SCT Test Levels/Phases

Test Level/Phase		Description
Component System Testing		Testing the complete CBS composed of multiple components to conform to system specifications and requirements. The testing examines and validates functions, performance, boundary conditions and other system properties.
Component Integration Testing		Testing multiple collaboration components that are integrated together to form heavy-weight complex components/subsystems. The inter-component testing examines integration architecture and design, interactions and relationships among integrated components in the component integration context. It is coarse-grained integration testing compared to class integration testing.
Single Component Testing	Component Class Integration Test	Testing a cluster of interdependent and coupled classes that are integrated together to form the CUT. The interclass testing examines multiple composite class interfaces and interactions across the collective class units in the integration context of the CUT. It is a foundation of high-level integration and system testing.
	Component Class Unit Testing	Testing a particular class unit that forms the CUT wholly/partially. The testing examines more fine-grained class operations than component operations, and tests all public and non-public operations of the class under test. Note that public class operations are candidates for constructing component operations, while non-public operation may not part of any component operations. A component class is a basic test unit in SCT.
	Component Operation Testing	Testing one or more component operations to exercise and examine a particular function or behavioural capability of the CUT for which the component operation fulfils wholly/partially. The testing may involve testing of several class operations (in the same class or from different classes) that jointly form the specific component operation under test.

In Table 2.7, the relationship between these SCT test levels is illustrated with a sequence of upward arrows, which indicate that the testing complexity increases as the test level ascends and vice versa. A lower level is usually regarded as a foundation for the next higher level. However, a test level may not be completely adequate in one way or another in testing practice. Accordingly, integration testing can detect and uncover component faults that are not only in the SCI context but also in the unit context.

SCI is a very common component reuse to produce complex components and CBS. While each component is tested individually in its development context, it must be also tested in the SCI context. The concept of CIT builds on the basic definition of general integration testing as follows: “testing in which software components are combined and tested to evaluate the interactions between them” [77]. A software component, whether it is integrated individually or with other components or modules, requires CIT to examine and ensure that component collaboration and interaction are correct in the actual SCI environment. In CBSE practice, the component user is concerned more about CIT, which can really determine whether a particular component is

selected and reused correctly in the user's CBS. CIT becomes an indispensable testing phase in the SCT domain.

Our MBSCT methodology has a principal focus on CIT and covers both inter-class integration testing and inter-component integration testing, which bridges component unit testing and system testing. The MBSCT methodology aims to detect and diagnose component faults particularly in the SCI context.

2.5 A Taxonomy of Software Component Testing Techniques

There are various SCT techniques, making it difficult to identify a common homogeneous basis to classify all testing techniques appropriately. As a key focus of SCT, test design and generation are based on component development information, such as component requirements, analysis and design specifications, component implementation (software programs or executable code), etc. On this basis, we can develop a useful taxonomy of SCT techniques, as shown in Table 2.8 (which is adapted from [165] [166]).

A major goal of this taxonomy is to classify commonly-used SCT techniques particularly describing approaches for test design and generation. The taxonomy also correlates the classified testing techniques to the relevant test levels. The first two types of testing approaches (IBT and SBT) represent the two main categories, where IBT typically supports unit testing, and SBT particularly supports integration testing and system testing. The last two types of testing techniques (MBT and UBT) fall into the sub-categories of the second main category (SBT). MBT and UBT are important testing techniques that will be comprehensively reviewed (in Chapter 3), and further developed and extensively applied in this research (from Chapter 4 and onwards).

Note that, although testing techniques vary with the testing information or artefacts used for test development, a key characteristic of SCT is dynamic testing (as described in Table 2.5) of software programs or executable code (which are the central subject of testing, as described in Section 2.2.6). Usually, software tests are not directly applied to or executed on software specifications/models, although these forms of "non-executable" software specification documents are a key foundation for fulfilling testing tasks (e.g. test design and generation, test result evaluation). Conversely, we view that "testing" of software specifications/models is *verification*, which is conducted "*indirectly*" or "*implicitly*" mainly through dynamic testing of their implementation (software programs or executable code). In particular, dynamic testing is undertaken with tests that are derived from software specifications/models and applied to software implementation in the runtime execution environment. If the dynamic testing results reveal some defects or imperfections in the software specifications/models, they can be rectified and improved to ensure that the software implementation is correct. This is a typical use of verification of software specifications/models for the overall testing purposes. In other words, software

specifications/models (non-dynamic) are verified if their corresponding software implementation (dynamic) is tested. This is a fundamental property of testing (especially the relationship between SBT/MBT and IBT), which will be further exploited in this research (from Chapter 3 onwards).

Table 2.8 Taxonomy of Software Component Testing Techniques

Technique	Description	Component Information	Test Level
Implementation-based testing (IBT)	(1) IBT focuses test design and generation on component implementation, which is software program in the form of source code that finally implements the CUT as the executable software. (2) Testing mainly examines program structure, internal mechanisms and artifacts. (3) Synonyms: structural testing, program-based testing, code-based testing, white-box testing.	Component implementation, programs or source code	Unit testing
Specification-based testing (SBT)	(1) SBT focuses test design and generation on the specification of component requirements, analysis and design, other than on how the component is implemented in some programming language or computer platform. (2) Testing mainly examines software functions and behaviors. (3) Synonyms: functional testing, behavioral testing, black-box testing.	Component requirements, analysis and design specifications	Integration testing, system testing
Model-based testing (MBT)	MBT bases testing tasks (including test design and generation, test result evaluation) on the software model of the CUT. MBT is an important form of SBT where the component specification is a model-based specification.	Model-based component specification, software models for component development and construction	Integration testing, system testing
UML-based testing (UBT)	UBT is a type of MBT where the software models used for MBT are constructed and specified with UML modeling (UML models).	UML-based component specifications, UML-based software models for component development and construction	Integration testing, system testing

2.6 Software Component Testability and Improvement Approaches

Our software component definition (as described earlier in Section 2.2.3) explicitly states that testability is a key advanced component characteristic, which can aid testing efforts to effectively support component reliability and quality, and reduce testing costs [24] [66]. Improving component testability is vital to enhance the testability of component-based software and sys-

tems, because their testability is essentially based on the testability of individual composite components [66].

In this section, we address basic concepts and principles of software component testability, and discuss important characteristics of component testability as a key foundation for the measurement of "good software component testability". After studying the general steps and testing approaches to improving component testability, we develop a practical taxonomy of testability improvement approaches and conduct a comparative study and discussion on these approaches. The content of this section is mainly based on the research work on component testability and improvement approaches in [173] [175] [176].

2.6.1 Software Component Testability

2.6.1.1 Testability Concept

In principle, the concept of s*oftware component testability* builds on the basic definition of general software testability as follows [77]: (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

This definition implies that testability is a measurable software quality indicator that denotes the ease of testing for conformance to certain testing requirements and objectives, such as test effectiveness, test coverage and test adequacy criteria. Accordingly, we can identify two important aspects of testability as follows [59]:

(a) The way in which a software system and its components are developed to enhance *test effectiveness*:

This aspect concerns the development of a software system and its components, which needs to incorporate test enhancements (e.g. with certain testing-support mechanisms and facilities) to assist the establishment of test criteria and performance of tests.

(b) Certain software requirements to achieve *test adequacy*:

This aspect concerns certain testable and measurable software requirements that can be used as a sufficient basis to devise and define achievable and adequate test criteria and performance of tests.

2.6.1.2 Testability Characteristics

Testability analysis is very useful to evaluate the quality of software testing to achieve the desired software reliability. Voas and Miller [151] view software testability as one of three pieces (software testability, software testing and formal verification) of the "software reliability puzzle" as they called it. To enable component functionality and reliability to be easily assessable and predictable, we can use the following five testability characteristics as a key foundation for the measurement of "good" software component testability: component traceability, component observability, component controllability, component understandability, and component test support capability. We can illustrate these component testability characteristics with a testability fishbone diagram as shown in Figure 2.2 (adapted from [173]).

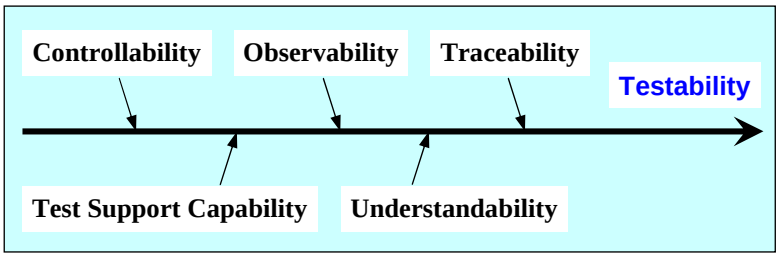


Figure 2.2 Characteristics of Software Component Testability

Among them, Freedman [59] uses observability and controllability to describe what he called "domain testability". Binder [23] also considers testability having these two key facets and discusses traceability for testability representation and test support environments. Gao et al. [64] particularly studies component traceability and tracking solutions. More recently, Gao et al. [66] [67] further discusses component testability in terms of these five testability properties.

The five characteristics of software component testability are described as follows:

(1) *Component Traceability* indicates how easy it is to track down different types of external /internal component behaviours and related program elements. Traceable components can facilitate and support tracing and recording specific component element information as necessary to reflect component execution information for component testing. The main component traces that can aid test effectiveness mainly include operation, state, event, error/exception, and performance traces.

(2) *Component Observability* indicates how easy it is to observe component testing information based on component operational behaviours, test inputs and actual test outputs for a particular test case. Well-defined component interfaces can enhance component observability to facilitate the establishment of the mapping relationship between test inputs

and corresponding test outputs. Observable component test artefacts aid the determination of how the given inputs affect the associated outputs during test execution. Component design and specification with enhanced component observability can support the monitoring of component functions and behaviours with associated component tests during the component development and testing process.

(3) *Component Controllability* indicates how easy it is to control component inputs/outputs, operations and behaviours of component execution during component testing. This property measures the ease of exercising component tests and producing a specific output in the output domain from a specific input in the input domain, so that certain expected outputs can be controllably predicted and produced from the associated inputs. Good component controllability can facilitate both development and verification of component tests.

(4) *Component Understandability* indicates how easy it is to understand component information, so that component testers can easily use/reuse relevant component information (e.g. requirements and specifications) for testing purposes, and design effective component tests and criteria for SCT. This characteristic involves two main aspects: (a) the *availability* of component information, i.e., how much component documentation is provided, such as component requirements, specifications, source code, user manuals, etc; (b) the *understandability* of component information, i.e. how well component information is presented in component documentation (e.g. being readable and understandable). Highly understandable components can improve test effectiveness and adequacy.

(5) *Component Test Support Capability* indicates how well component test automation is supported with capable software tools. This characteristic particularly focuses on test operation during testing, and involves four main aspects: test generation capability, test management capability (e.g. to manage test cases, test process, etc.), test coverage analysis and evaluation capability, and test execution and support capability. Well-supported test automation can improve test effectiveness and efficiency.

The first three characteristics are very important for providing good component testability. Technically, component traceability is an essential property that affects and supports component observability and controllability. Strong component testability can reinforce component design and specification to be able to trace, observe and control component behaviours and test elements (e.g. operation, state, event, etc.) of component execution for component testing, in order to facilitate the establishment of appropriate test criteria to enhance test effectiveness and efficiency. This research seeks useful test mechanisms and techniques to improve component

testability with a particular focus on the enhancement of the first three testability characteristics described above.

2.6.2 General Strategies to Improve Component Testability

In practice, component developers and testers often encounter some critical questions during design/testing phases:

- How to improve component testability?
- How to develop testable components?
- How to facilitate SCT activities for good component testability in an effective and systematic way?

To address these questions, this section examines general strategies (including general steps and testing approaches) to improve component testability. We then present our taxonomy of testability improvement approaches and conduct a comparative study from different perspectives.

2.6.2.1 General Steps to Improve Component Testability

With regard to what component development steps are associated with testability improvement, there are two main steps:

(a) During the SCD process: component developers need to apply appropriate testing techniques to design testable artefacts, and incorporate testability enhancements together with component design and specification. Such testability improvement before testing is conducted supports component test design effectively during the testing step. This approach is in line with test-driven development [15] [79].

(b) During the SCT process: if component testability is not considered or insufficiently applied in the SCD stage, component testers will have to subsequently apply certain testing techniques to enhance component design and specification for component testability. Such post-design testability improvement is necessary before component test development.

The first step described in (a) above is strongly recommended, which can alleviate substantial testing overheads in the later testing stages. The second step described in (b) above is also used, although the workload of testability enhancements may vary in SCT practice. In many situations, testability improvements are often undertaken in both steps in CBSE practice.

2.6.2.2 A Taxonomy of Testability Improvement Approaches

With regard to general testing methods in the literature, there are certain testing approaches particularly for component developers/testers to incorporate appropriate testing-support artefacts (e.g. assertions [164] [151] [152] [123] [153]) for improving component testability. We can develop a practical taxonomy that contains four main testability improvement approaches, as described in Table 2.9 (Taxonomy Part 1).

Table 2.9 Taxonomy of Testability Improvement Approaches (Taxonomy Part 1)

No.	Approach	Description
#1	Framework-based testing facility [81]	This approach develops a well-defined testing framework (e.g. testing-support class libraries and tools) that is dedicated to facilitate testability improvements. Component testers can use the testing framework to add in the component program appropriate test code that accesses test interfaces of the test framework and interacts with the framework's testing-support tools. As a typical example, JUnit is a lightweight testing framework that supports adding simple test code typically for unit testing of Java class code.
#2	Built-in tests [157] [158] [159] [12]	This approach allows component developers/testers to add or embed built-in tests (e.g. assertions) as extra (non-functional) component code artefacts along with component implementation, and supports self-checking and self-testing at runtime. Built-in tests are usually not part of the original component functional requirements, and they are added especially for the testing-support purpose.
#3	Component test wrapping [65] [17] [56]	This approach aims to augment and convert a basic component to be a testable component by means of wrapping the corresponding CUT mainly with additional testing-support artefacts, and to produce a companion component test wrapper to facilitate component testing. Being separate from the CUT, the companion component test wrapper is executable, deployable and testable particularly for testing of the CUT and its related interacting components.
#4	Component test bench and provider certification [96] [98]	This approach requires that component providers package software components with executable CTCs and test results (e.g. stored in XML documents), and accompanying testing-support tools, which have all been developed for component testing and certification. Component users can directly perform component verification and validation with the provided CTCs and tools for re-testing in the final application environments. With component providers taking the main testing responsibilities to greatly reduce testing costs for component users, this approach is provider self-testing and self-certification, and offers verifiable testability evidence to component users immediately.

In developing this taxonomy, we study the main features of these approaches from different perspectives and conduct relevant comparisons, as described in Table 2.10 (Taxonomy Part

2), which is adapted from [173] and extends a similar description in [66] with appropriate enhancements as indicated (especially for Approach #4). A further comparative study based on this taxonomy is presented in the next section.

Table 2.10 Features and Comparisons of Testability Improvement Approaches (Taxonomy Part 2)

Different Perspectives	Framework-based testing facility	Built-in tests	Component test wrapping	Component test bench and provider certification
Developer/provider-oriented	Yes	Yes	Yes	Yes
User-oriented	No	No	Yes	No
Component interface access	Yes	Yes	Yes, access only	Yes
Component source code access	Yes	Yes	No	Possible, if needed
Test artefacts are code-embedded in CUT	Yes	Yes	No	Possible, if needed
Wrapper: test artefacts separate from CUT	No	No	Yes	Possible, if needed
Overheads for test programming	Low	High	Low	Low/High
CUT complexity with test artefacts	Low	High	Low	Low/High
Test change impact on CUT	No or Low	Yes & High	No or Low	No/Yes
CUT change impact on component tests	No or Low	Yes & High	No or Low	No/Yes
Usage flexibility (approach usability)	High for providers	Low for providers	High for providers/users	High for users
Test level	Focusing on unit testing	Focusing on unit testing	All test levels	All test levels
Applicable component types	In-house & newly developed components, in-house legacy modules			
			COTS	

2.6.2.3 Comparative Study

In Table 2.10 (Taxonomy Part 2), we showed the main comparisons of testability improvement approaches from different relevant perspectives. The following discussion describes a comparative study of these four main approaches.

(1) The first and second approaches both mainly work on increasing code-based testability, which is not very suitable for higher-test levels, such as component functional testing and integration testing based on design specifications and models. Both approaches assume that component source code is accessible, which is also unsuitable for SCT activities conducted by component users who do not have the same privilege of component source code access as component developers or in-house component testers have. In addition, inserting built-in test code into component code could lead to inadvertently incorrect component changes, which may negatively impact testing.

(2) By contrast, the third approach has advantages over the first two approaches, and is more flexible for improving code-based and design-based component testability. This approach can enhance component testing capabilities for component functional testing and integration testing mainly via component interfaces and specification, without access to the low-level details inside component code. Another advantage is that a component test wrapper can be developed and implemented as well as executed in parallel in the same way as its counterpart CUT, especially by using the same design approach and programming language, and by executing in the same runtime environment.

(3) The fourth approach takes a very different SCT strategy from the other three approaches. This approach shifts almost all testing effort to the component provider side, and thus greatly reduces testing costs for component users, although the actual testing effectiveness and quality may depend on the CTCs and testing tools provided by component providers. As the production-side testers have the privilege of accessing all component specifications and source code as well as the internal technical support, this approach could produce high quality component tests in a more effective and efficient way than the same work conducted by the user-side testers. In practice, component providers can employ the first three approaches and other SCT techniques to develop CTCs and testing tools, and fulfil the provider certification required in the fourth approach.

(4) From the viewpoint of component stakeholders, all the four approaches are provider-oriented, and only the third approach is user-oriented. A key advantage of the third approach is that the user-side testers can undertake SCT activities mainly via component interfaces and specification, without the prerequisite of component code access. The third approach is applicable to both component providers and users, so that they all can beneficially use a consistent approach to improve testability for desired test effectiveness.

2.7 Summary and Discussion

This chapter has provided a comprehensive review of the important concepts, principles, characteristics and techniques of software components (which are the primary subject of SUT in this research scope) and SCT (which is the central focus of this research). We identified the special CBSE diversity characteristic (as described earlier in Section 2.1 and Section 2.2.1.4) as a principal study theme. Based on this, we discussed a number of important issues (as introduced earlier in Section 2.1) concerning software component technology for SCD and SCT.

Furthermore, we have carried out further research work that extends and consolidates the relevant research foundation, and accomplished a number of research findings and results (including new definitions and concepts), which are summarised below with our intention about how to apply them in this research:

1. A new comprehensive taxonomy of software component characteristics (in Section 2.2.2)

This new comprehensive taxonomy of software component characteristics has been systematically developed, based on our study of component concepts and characteristics. This taxonomy classifies twenty-two (22) software component properties into four (4) main categories at different componentisation levels, and shows characteristic interrelationships between component properties as well as component stakeholders. Seven (7) new component characteristics have been identified and added to the taxonomy to emphasise high-level component properties. This work enables this taxonomy to be more informative and comprehensive than the existing component characteristic classifications in the current literature.

2. A new software component definition (in Section 2.2.3)

This new software component definition was based on our new taxonomy of software component characteristics. Compared with other component definitions in the literature, our component definition extends the definition converge scope by adding new component quality properties (e.g. testability and reliability) to assure component functionality, reusability and other important component properties. This new component definition appears to be the most comprehensive available in the current literature in terms of the range of important component characteristics covered. This research applies this new component definition to develop the new MBSCT methodology to improve component testability and quality.

3. A new definition of software component testing (in Section 2.3)

This new SCT definition describes a generic testing process and the main testing tasks, which goes beyond the traditional scope of SCT at the individual unit or single component level. Based on the new SCT definition, we studied and analysed important SCT characteristics, test cases and specification concepts, and different testing perspectives and needs. This research focuses on CIT and integrates this new SCT definition to develop the new MBSCT methodology.

4. A useful taxonomy of software component testing techniques (in Section 2.5)

This taxonomy of SCT techniques was developed in terms of component development information used for component test design and generation. The taxonomy illustrates the relationship between the classified testing techniques and test levels. With support from this taxonomy, this research focuses on model-based testing for the goal of component integration and system testing.

5. A practical taxonomy of component testability improvement approaches (in Section 2.6)

Based on our study of component testability concepts, characteristics and improvement approaches, this practical taxonomy of component testability improvement approaches has been developed, in conjunction with a comparative study from different stakeholder perspectives. This research puts a particular emphasis on component testability improvement to support component quality and to achieve component testing effectiveness. The new MBSCT methodology aims to improve component testability and quality.

The comprehensive literature review and further research results in this chapter make original contributions to the body of knowledge in the main research areas of SCD and SCT in the literature. This has created a key conceptual foundation to support the development of the new MBSCT methodology.

Chapter 3
Foundation of Model-Based Testing and UML-Based Testing

3.1 Introduction

Model-based testing (MBT) emerges as a natural advancement of specification-based testing (SBT), where software models are used as model-based specifications for software testing. MBT is a new and evolving testing paradigm, and possesses its own concepts and features different from traditional testing techniques (to be described in Section 3.2). MBT has been becoming increasingly popular and is now a mainstream software testing approach, especially MBT with UML, namely UML-based testing (UBT). This is mainly due to the popularisation of emerging model-centric software development paradigms and their intuitive connections to MBT, such as the standardised UML (Unified Modeling Language) [108] [28] [125] and MDA/MDD/MDE (model driven architecture/development/engineering) [106] [84] [134].

MBT/UBT is the primary software testing approach we use in this research. Among many other modeling and testing aspects, this chapter focuses on a number of important issues and challenges in the principal areas of MBT and UBT:

(1) What is model-based testing? Why model-based testing? (in Sections 3.2.1 and 3.2.2)

(2) What testing tasks can be model-based? (in Section 3.2.3) What are model-based tests? (in Section 3.2.4)

(3) How do we develop a new MBT definition to reinforce the integration of MBT with MBD into the entire SDLC process? (in Section 3.2.5) What are main MBT advantages? (in Section 3.2.7)

(4) What types of models can be used for MBT? What is a test model? What is a good strategy to obtain test models? (in Section 3.2.6)

(5) What is UML-based testing (UBT)? How do UML models fit into MBT? (in Section 3.3)

(6) What are the main aspects of software integration testing with UML? (in Section 3.3.2 and Section 3.4)

(7) What is use case driven testing? (in Section 3.3.3) What are the main aspects of software system testing with UML? (in Section 3.3.2 and Section 3.4)

(8) What are the main outstanding problems and limitations in MBT/UBT? (in Section 3.5)

This chapter presents a comprehensive review of important concepts, principles, characteristics and techniques of MBT in general and UBT in particular, which aims to create a solid technical foundation in these important research areas to support the new MBSCT methodology that is proposed and developed by this research. We study and review related research work on MBT/UBT in the literature, and identify and analyse the main problems and limitations in the current MBT/UBT domain. At the same time, we undertake further research work to develop new concepts and definitions, with the intention of enhancing the relevant knowledge and principles of MBT/UBT in the literature.

The remainder of this chapter is structured to cover the abovementioned important issues in MBT/UBT. Section 3.2 reviews important MBT concepts, principles, characteristics and associated issues. We propose a new MBT definition (in Section 3.2.5) and a new test model definition (in Section 3.2.6) based on our further research work. In Section 3.3, we propose a new UBT definition (in Section 3.3.1), and describe main UBT concepts and associated issues particularly on how UML models support MBT. In Section 3.4, we comprehensively review research work related to state-based testing (in Section 3.4.1), software integration testing with UML (in Section 3.4.2), software system testing with UML (in Section 3.4.3), software testing with UML use cases and scenarios (in Section 3.4.4), and software testing with UML sequence diagrams (in Section 3.4.5). Section 3.5 examines the main problems and limitations in MBT/UBT. Finally, Section 3.6 presents a summary of this chapter. A more detailed literature review of MBD/UML and MBT/UBT can be found in [177].

3.2 Model-Based Testing

3.2.1 What is Model-Based Testing?

The idea of MBT originates from MBD, and both share common concepts and characteristics of model-based approaches. Intuitively, *model-based testing* is a general term denoting that software testing is based on software models of the SUT. MBT derives test cases from software models, not from source code. As software models describe software requirements and functional specifications, MBT is usually regarded as a form of black-box functional testing. MBT generates functional tests that can be applied to all test levels and that are more effective for integration testing and system testing.

There are many types of testing techniques (using certain models) developed by academic researchers and industry practitioners with different testing views, which leads to the situation that there is no single formal MBT definition that has been well accepted and widely used by all. Table 3.1 summarises some of the existing MBT definitions in the literature.

Table 3.1 Review of MBT Definitions

Definition Reference Source	Definition Description
Definition by Dalal et al. [46]	According to Dalal et al., model-based testing means an approach to automatic test generation using models extracted from software artefacts.
Definition by El-Far & Whittaker [57]	According to El-Far & Whittaker, "model-based testing is a general term that signifies an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the application under test."
Definition by Pretschner et al. [118] [120]	According to Pretschner & Philipps, "the idea of model-based testing is to use explicit behaviour models to encode the intended behaviour and to derive test cases that are used for verifying the respective implementation."
Definition by Gross [69]	According to Gross, "model-based testing is the development of testing artefacts on the basis of UML models, which provide the primary information for developing test cases and test suites, and for checking the final implementation of a system."
Definition by Utting & Legeard [148] [150]	According to Utting & Legeard, "model-based testing is the automation of the design of black-box tests."
Definition by Frantzen & Tretmans [62] [141]	According to Frantzen & Tretmans, "in model-based testing, a model of the desired behaviour of the implementation under test is the starting point for test generation and serves as the oracle for test result analysis."
Definition by Hartman et al. [71]	According to Hartman et al., "in model-based testing, tests are generated automatically from models that describe the behaviour of the system under test from a perspective of testing."
Definition by Bertolino [21]	According to Bertolino, "the leading idea of model-based testing is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases."
Definition by Pezze & Young [112]	According to Pezze & Young, "model-based testing consists in using or deriving models of expected behaviour to produce test case specifications that can reveal discrepancies between actual program behaviour and the model."

From a review of each of these definitions, we can see that most of the existing MBT definitions are given informally in certain contexts, and develop some specific testing characteristics and/or purposes. In the case of MBT, the target of testing remains unchanged, which means that MBT aims to test the implementation of the SUT as a key testing goal shared by all testing approaches. However, the basis of testing for MBT shifts to models, not based on implementation/code or some other basis, compared to traditional testing paradigms. Accordingly, the principles of MBT should reflect relevant model-based implications for effective software testing, in terms of important MBT-related concepts and characteristics.

3.2.2 Why Should Testing Be Model Based?

A primary reason why testing should be model based is that software models capture system requirements and functionalities that determine the aspects of both software design and testing. In the case of use case driven development, use case models are used throughout software analysis, design, implementation and testing. Another reason is that MBT could take advantage of good principles and characteristics of MBD. One of the fundamental MBT principles is that applying software models to software design and software testing enables both phases to utilise a consistent model-based specification approach to producing functional and reliable software with better effectiveness and efficiency.

In the common context of MBD, software models are constructed usually before or parallel to the actual development of the SUT, and naturally become a central foundation of software testing. As a quick overview, we examine two typical usage situations, where MBT is especially suitable:

(a) For a new system under development and test

For a new system, MBT enables testing to start before coding, which is a key advantage of MBT (in Section 3.2.7). In this situation, since the system development is not finished, software models are the only source of testing information available for undertaking testing tasks.

(b) For a developed system under test

Another situation is that the system has been developed from software models, but it has not been tested yet or it needs further testing. In this situation, because software models capture system requirements and software development information of the SUT, they naturally become a better choice as a testing basis to examine and evaluate the SUT.

MBT is a representative paradigm of SBT. Compared to traditional IBT [16] [100], SBT has more advantages and benefits, as shown in several studies [24] [104] [105] [165]. In particular, Binder indicates that traditional IBT has “substantial limitations”, and “should not be the primary basis for testing” [24]. Section 3.2.7 further discusses a number of MBT advantages and benefits to demonstrate that MBT is very suitable and widely used in the software testing domain.

3.2.3 What Testing Activities/Tasks Can Be Model Based?

In common with MBD that bases common development tasks on software models, MBT supports important model-based software testing activities and tasks, which are summarised as follows:

(a) *Test Analysis* begins with a model of the SUT, analyses model artefacts describing the system behaviour under test, and explores test strategies to examine the respective SUT behaviour. Model-based test analysis serves as a starting point for subsequent model-based test design and generation.

(b) *Test Design and Generation* develops test cases based on the SUT model in accordance with specified test strategies and/or testing objectives. In particular, certain testable model artefacts are extracted from the SUT model, and are further transformed (with possible test improvement) into test data to produce and represent test cases (called *model-based tests*).

(c) *Fault detection* reveals possible software faults with model-based tests against the expected SUT behaviour captured by model-based requirements and specifications.

(d) *Test Evaluation* assesses software correctness and quality of the SUT against model-based requirements and specifications as well as target testing objectives.

3.2.4 Model-Based Tests

MBT derives model-based tests from software models of the SUT, which is performed manually or by certain testing tools. There is a question raised here: are model-based tests executable on the SUT for dynamic testing? Technically, there are two steps required to develop model-based tests for dynamic testing of the SUT.

(a) Step #1: abstract test cases

A model usually shows a part of the SUT behaviour, because by nature it is only a simplified representation of the SUT at a certain level of abstraction or precision. Accordingly, test cases developed directly from the model remain at the same level of abstraction as the model, and are originally represented in terms of abstract data and operations extracted from the model. Thus, at least at the initial stage, such model-based tests are usually regarded as *abstract test cases*. Because models and code appear at different levels of the SUT, these abstract test cases are not directly executable against the SUT, while tests derived from code usually can be executed on the SUT. This means that the initial abstract test cases derived directly from an "abstract" model of the SUT are not ready to be used for the dynamic testing of the SUT.

(b) Step #2: concrete/executable test cases

Dynamic testing requires test cases to be executed on the concrete implementation of the SUT. For this testing purpose, it is necessary for MBT to undertake a further test development

step: mapping and transforming the abstract test cases derived from a model of the SUT into low-level concrete test cases that are ultimately suitable for test execution in dynamic testing of the SUT. Such test mapping and transformation steps are an important part of the MBT process, with the aim to make model-based tests derived from the SUT model executable on the SUT implementation for dynamic testing. This research addresses this important MBT issue concerning test mapping and transformation with the development of the new MBSCT methodology.

3.2.5 A New Definition of Model-Based Testing

Based on our review of existing MBT definitions (in Section 3.2.1), and of important MBT-related issues and characteristic aspects (in Section 3.2.2 to Section 3.2.4), we propose a new definition of model-based testing in this research as follows:

> **Definition 3–1.** *Model-based testing* bases software testing on explicit software models with model-based development of the software/system under test. In the model-based testing process, MBT particularly designs and generates test cases (with oracles), and evaluates test results based on the relevant software models and model-based specifications for testing the SUT.

In the following, we discuss some important implications associated with our MBT definition, in comparison with other existing MBT definitions (as reviewed in Section 3.2.1):

(1) A distinguishing feature of our definition is that this definition firstly emphasises the intrinsic connection of MBT to its counterpart MBD, which is a key difference from the other existing MBT definitions. Both MBT and MBD should be integrated and collaboratively work together in the iterative/incremental software development process (this point is further amplified in Section 3.2.5.1 below).

(2) This definition emphasises that the testing basis is explicit software models and model-based specifications that describe and represent the SUT on which MBT undertakes the model-based software testing process.

(3) This definition contains important model-based software testing activities and tasks in the testing process, including test design, test generation and test evaluation.

(4) This definition emphasises and supports the general testing goal: MBT aims to test the implementation of the SUT by using test cases that are derived from model-based specifications.

3.2.5.1 Integrating MBT into the Entire Software Development Process

This section further discusses the importance of integrating MBT into the entire software development process, as emphasised by our proposed MBT definition above. For the purpose of effective MBT practice, we argue that MBT should not be simply based on a single unconnected model or some unsystematically-developed individual models that are not well connected to the current MBD process. As discussed in Section 3.2.6 below, not using any development models for MBT not only is unrealistic, but also wastes software development resources, and we should adapt relevant selected development models for MBT. We also argue that the importance of models constructed for software testing (i.e. *test models* as defined in Section 3.2.6) should be treated equally with models constructed for software development. We recommend that test models in MBT should be built in parallel to relevant development models in MBD. The effectiveness of the MBT process relies on the clear connection and close collaboration with the corresponding MBD process, where relevant software models have been designed and constructed to provide a solid foundation for different testing aspects and purposes. Based on the relevant MBD phases and associated development models, MBT can then take advantage of fully integrated approaches for collaboratively undertaking software modeling and testing. Both MBT and MBD should work together to fit into the entire SDLC process, in order to produce quality software effectively and efficiently.

This research incorporates our proposed MBT definition to develop the new MBSCT methodology with an iterative and incremental process of UML-based software component development and testing. This aspect is further discussed in Section 3.5.

3.2.6 Test Models

3.2.6.1 What Types of Models Can Be Used?

Software models used for MBT may appear in different types (e.g. process model, domain model, behavioural model, etc.), and can be represented in different modeling notations and/or languages (e.g. UML) [71]. There is no single model that is sufficient or perfect to solve all testing issues, and not all “models” are suitable for testing.

Different types of models may support different testing aspects or purposes. For example, process models are very useful to describe relevant testing processes for undertaking testing activities and tasks. Behavioural models specify important requirements and specifications for the system behaviour, which forms a MBT basis for model-based test analysis, test design and gen-

eration, and test evaluation. Compared to other types of models, an appropriate behavioural model can be enhanced to capture the expected behaviour of the SUT and describe important testing relationships between test inputs and outputs, so that the behavioural model can particularly support the derivation of test cases with *oracle* information (e.g. the expected test results of the SUT). This research mainly employs behavioural and process models in model-based testing of software components.

3.2.6.2 A New Test Model Definition

MBT conducts test derivation and evaluation based on software models. Informally, a model is referred to as a *test model* if the model is used in a MBT process. Based on our MBT definition (in Section 3.2.5), we propose a new test model definition in this research as follows:

> **Definition 3–2.** A *test model* denotes a test representation of the SUT in terms of models that describe the test relationships among elements of the SUT.
>
> Model-based testing constructs test models and applies them to undertake testing activities, especially model-based test design, generation and evaluation.

There is a close relationship between test models and MBT. MBT starts with test model development, which is the first important testing task in MBT. As indicated in Section 3.2.6.1 above, not all "models" are suitable for testing, and if a model is not test-ready or non-testable, it cannot be used directly for MBT. For the purpose of desired MBT effectiveness, test models must be developed to be test-ready and testable to support important model-based testing tasks (as described in Section 3.2.3).

3.2.6.3 Bridging "Test Gaps"

There are some questions with regard to test models for MBT: where do test models come from for MBT? How are test models in MBT different from existing software models (e.g. design models) in MBD? What is a good strategy to obtain test models? To answer these questions, we examine the following three main approaches to obtain a test model for MBT [150]:

(1) Fully reusing a simple or ordinary software model directly from software development as a test model with no modification

The full "as is" reuse of a simple or ordinary software model without any change is usu-

ally not applicable in MBT practice. A key reason is that there exist certain "*test gaps*" between ordinary software models (which are not test-ready or non-testable) and target test models (which are test-ready or testable), because such ordinary software development models tend to focus on software design/implementation, and often may not contain adequate testing-related information and testing-support artefacts required for effective test generation in MBT. Such "test gaps" are a major cause of inadequate model-based testability in MBT. This is not an approach that would usually be used in MBT practice.

(2) Building a test model exclusively for MBT from scratch without using existing software development models

No reuse of any existing software models is impractical in MBT practice. On the one hand, this approach would simply ignore available software information described by software models (e.g. the SUT functions) that is implicitly/explicitly useful for testing, which wastes software development resources and accordingly causes testing to be more costly. On the other hand, any test model would eventually contain some software artefacts already described in relevant software models of the SUT, which means this approach is unrealistic. Accordingly, this is not an approach that would usually be adopted in MBT practice.

(3) Transforming and improving a selected development model (e.g. a design model) into a target test model

Based on the analysis of the rejected approaches above, it is necessary for MBT to take an intermediate approach that appropriately reuses and adapts selected software models (e.g. design models) as a key testing basis for developing target test models (e.g. design test models). The MBT tester needs to undertake additional "*remodeling*" activities to transform and improve non-testable models into testable models, before the models can be used for MBT. In particular, the MBT tester needs to bridge the identified "test gaps" for model-based testability improvement in test model development. This is an integrated approach, which is practical and cost-effective in MBT practice.

In MBT practice, a test model is not exactly the same as its associated development model that is selected from MBD for test model construction. In some cases, a test model could be smaller and/or simpler than its associated development model, in terms of software details contained for a particular testing purpose. A "rule of thumb" is that a good test model should be reasonably simple and/or more abstract than the concrete implementation of the SUT, but it also must be adequately precise for the target testing objective [116] [119] [149] [150].

In this research, we introduce this new notion of "test gaps" to emphasise a major MBT focus: remodeling and improving model-based testability to bridge the identified "test gaps" for

effective test model construction. This is a starting point that has motivated this research to adopt the abovementioned integrated approach (see (3) above), and to develop the *Test-Centric Remodeling* strategy and the *Test by Contract* technique with the new MBSCT methodology. This aspect is further discussed in Section 3.5.

3.2.7 MBT Advantages and Limitations

MBT has become a mainstream testing approach nowadays [24] [150]. While borrowing or inheriting some features from the general model-based paradigms, MBT retains its own testing principles and characteristics, and holds various benefits and advantages, compared to traditional testing approaches, such as code-based testing, white-box testing, or other manual testing approaches (without using models) (e.g. manual test design, hand-crafted individual tests).

The following summarises a number of the important MBT benefits and advantages (in terms of overall MBT, but not related to specific MBT techniques):

(1) Model-based requirements and specifications capture and simulate actual functions, behaviours and scenarios of the SUT
- Using explicit models helps understand the SUT and clarify the requirements
- Provide a key basis for test design, generation and evaluation
- Direct testing towards the correct starting point and direction, i.e. go for MBT

(2) Effectively support black-box functional testing with the aid of model-based requirements and specifications.

(3) Virtually support all test levels, and more effectively on integration and system testing.

(4) Make model-based test cases independent of the implementation of the SUT, with no assumptions on particular implementation aspects and/or internal structures
- Support test cases to be reusable
- Particularly benefit SCT in different component implementation/application contexts, i.e. develop model-based component cases independent of component implementation to support special SCT needs

(5) Enable test case development to get started much earlier in the SDLC, so that test cases are ready for test execution before the SUT implementation is started or completed
- Shifting testing earlier than coding
- Shifting testing earlier for effective test plan and test development

- Reduce testing time and costs

(6) Help identify system errors/faults (deficiencies, inconsistencies and/or incompleteness) in requirements/specifications in the earlier phases of the SDLC
   - Improve the requirements/specifications before the implementation starts
   - Reduce/save development time and resources

(7) MBT studies show that model-based tests (which are automatically or manually derived from explicit model-based requirements) detect more requirement-level errors than manually designed tests (or hand-crafted individual tests) directly from informal requirements [121] [46].

(8) Provide a potential for automated testing with the aid of model-based testing tools.

However, MBT also has certain disadvantages and limitations, for example:

(a) Possibly miss some faults, because the model is not exactly the same as the low-level concrete implementation of the SUT. That is, complete or exhaustive testing with MBT is still unachievable (as described in Section 2.3.5).

(b) Difficult to measure the quality of model-based tests to achieve high testing coverage, because of basic model characteristics: abstraction and simplification.

(c) Require more knowledge and skills of both modeling and testing for testers than traditional testing approaches (e.g., code-based testing, manual testing without models).

3.3 UML-Based Testing

Model-based testing with UML or UML-based testing (UBT) emerges as a new approach to MBT. UBT is the major type of MBT approach we use in this research.

3.3.1 A New Definition of UML-Based Testing

Technically, UBT is a new type of MBT where software models used for testing are UML-based software models (or UML models for abbreviation) that are developed with UML diagrams and specifications. We propose a new UBT definition that is derived from our earlier MBT definition (in Section 3.2.5) as follows:

Definition 3–3. *UML-based testing* bases software testing on explicit UML-based software models with UML-based development of the software/system under test. In the UML-based testing process, UBT particularly designs and generates test cases (with oracles), and evaluates test results based on the relevant UML-based software models and UML-based specifications for testing the SUT.

In principle, UBT retains the important concepts, principles and characteristics of MBT (as described in Section 3.2). Moreover, UBT has its own testing features and capabilities that benefit from the standardised notations and rigorous semantics of the UML. One promising benefit is that the UML enables software testers to employ standard modeling notations rather than non-standard ones, and take advantage of useful UML features in MBT activities. Another benefit is that, by unifying UML-based development and testing together, software engineers can utilise a consistent UML-based approach and specification for both component development and testing to produce functional and quality software components and systems with better effectiveness and efficiency. Therefore, UBT really fits into MBT practice, and advances MBT a further step.

3.3.2 UML–SCT: A Core UML Subset for SCT

The UML is very complex and contains a comprehensive set of numerous modeling diagrams and notations for general-purpose system modeling. The current UML 2.x defines 13 types of modeling diagrams. In practice, software engineers often need to select and use a subset of the UML that is most suitable and useful for their practical development purposes. Fowler [60] indicates that class diagrams and sequence diagrams are the most common and useful types of UML diagrams. Dobing & Parsons [55] review the UML literature, and survey the UML practitioners and their clients. Their research results show that class diagrams, sequence diagrams and use case diagrams are used most often, while communication diagrams are used least (note that communication diagrams in UML 2.x were called collaboration diagrams in UML 1.x). Dias Neto et al have recently conducted a more comprehensive survey of MBT/UBT approaches in the literature [47] [48] [49] [50] [51] [52] [53]. One of their findings is that, among different types of behavioural models in all 47 analysed papers using UML, the top 4 of the most used UML diagrams are statechart diagrams, class diagrams, sequence diagrams, and use case diagrams [47].

For the goal of UML-based component development and testing in this research, we select and use a core UML subset (called *UML–SCT*), which mainly includes use case diagrams, activity diagrams, class diagrams, sequence diagrams, and statechart diagrams, as well as OCL

expressions [160]. The five main UML diagrams, notations and semantics in UML–SCT are the same as defined in the standard UML 2.x. The literature review (as summarised in the above paragraph) shows that our selection of UML–SCT is well consistent with the commonly-used UML diagrams to support UML-based SCT.

Table 3.2 summaries UML diagrams and modeling for software testing, which focuses on the five UML diagrams in UML–SCT. There are some important implications for the use of UML diagrams in testing:

(1) For requirements-based system testing, we employ UML use case diagrams, sequence diagrams and class diagrams.
(2) For integration testing, we also use the above UML diagrams at the integration level.
(3) For unit testing, we mainly use UML state diagrams and class diagrams.
(4) Testing (at any level) must use relevant UML diagram descriptions and model specifications, because a graphical diagram alone provides only very limited information for testing [69].

Table 3.2 UML Diagrams and Modeling for Software Testing

View/Type	Diagram	Modeling for Testing	Test Level
Requirements	Use Case Diagram	Modeling system requirements with use cases, actors, and their interaction relationships, system behaviour and events. Deriving system/integration test requirements, high-level test design.	Integration /System Testing
	Activity Diagram	Modeling a process/workflow of control and data computation step by step, dynamic system behaviour and procedural/parallel functions. Complementing test requirements.	Integration /System Testing
Behavioural /Dynamic	Sequence Diagram	Modeling a sequence of temporally-ordered messages (method calls) for dynamic interactions between objects in realising use case scenarios. Deriving test scenarios, test sequences of test sets.	Integration /System Testing
	State Machine Diagram	Modeling states and their transitions for event-ordered dynamic behaviour of an object. Deriving unit tests.	Unit Testing
Structural /Static	Class Diagram	Modeling classes (attributes and operations), interfaces and their relationships for the static design structure of a system.	Unit Testing supporting all test levels

In our MBSCT methodology, we mainly employ use case diagrams (use case view), sequence diagrams (behavioural/dynamic view) and class diagrams (structural/static view) in UML–SCT to undertake UML-based SCT for component test design, generation and evaluation. The following subsections (Sections 3.3.2.1 to 3.3.2.3) further discuss these three most often used UML diagrams in UML–SCT and how they are used to support MBT activities.

3.3.2.1 UML Use Case Diagrams for Software Testing

UML use case diagrams can model the requirements of systems, subsystems and integration functions (i.e. what a system/subsystem should do). Use case diagrams can show many aspects of system requirements, and use case specifications describe functional behaviour and requirements, allocation of functionality to classes, object interactions and object interfaces, user interfaces, and user documentation [24]. Use case diagrams and associated use case specifications are a suitable and important resource for deriving system/integration tests for UML-based system/integration testing.

The following gives a general process for deriving test cases from use cases:

(1) Step #1: Developing use case instances or scenarios

The tester needs to analyse each use case and identify all use case instances or scenarios, including success, variation/alternative, failure, and exception scenarios.

(2) Step #2: Developing abstract test cases

The tester needs to identify and design at least one test case for each use case scenario, especially developing test cases for core scenarios in the use case. These model-based tests are initial test requirements and/or abstract test cases at high-level test design (in Section 3.2.4).

(3) Step #3: Developing concrete/executable test cases

The tester needs to construct actual test data and transform abstract test cases to generate concrete test cases suitable for test execution for dynamic testing of component implementation.

(4) In principle, the basic test coverage required is that test cases at least cover each use case and each actor in the use case diagram.

From the above test derivation process, we can see that the first two steps are mainly based on UML use case diagrams. However, after developing abstract test cases with use cases, the tester needs to employ other UML diagrams and/or testing-support information to identify and construct the necessary test data to generate individual tests for developing concrete test cases.

3.3.2.2 UML Sequence Diagrams for Software Testing

UML sequence diagrams mainly focus on realising (functional and operational) scenarios of use cases in the use case model, where use cases need to be refined into one or more sequence dia-

grams for the next level of refinement and development. Sequence diagrams show dynamic modeling of system behaviour, which is specified by the associated use case scenarios, and the sequentially time-ordered interaction messages (in the form of method calls/invocations) between participating objects that communicate and interact with each other, and collaborate to accomplish some tasks or functions in the integration or subsystem/system context. Accordingly, UML sequence diagrams are mainly used as a basis to derive test sequences (consisting of test messages or test operations), which can drive design and generation of integration/system tests for software integration/system testing. In principle, a sequence diagram corresponds to one or more test sequences and test cases that have different states of interacting objects in software integration.

3.3.2.3 UML Class Diagrams for Software Testing

UML class diagrams represent the static structure of a system, and show how the system is structured or organised, rather than how the system behaves. Class diagrams define and show the system's classes, attributes, operations, interfaces, and their static structural relationships, which focus on how classes are related but not how classes interact with each other. Class diagrams specify these important model elements and software artefacts that can be used for describing various software models and specifications (e.g. object-oriented analysis and design models), and for constructing software implementations through forward and reverse engineering. Class diagrams define what software classes are needed to capture the interaction relationships between all objects participating in sequence diagrams, and what class operations and attributes are needed in relevant software classes to implement and represent the required functions. As the most common UML diagrams used in object-oriented modeling, class diagrams provide important test information to construct basic test data for deriving concrete test sequences and test cases, which can used to support testing at all test levels.

3.3.3 Use Case Driven Testing

Use case diagrams are usually considered as the starting point for test case design and generation, particularly for component functional testing at all test levels. This is where the idea of a use case driven testing approach comes from. By using UML use case diagrams for software testing as described in Section 3.3.2.1 above, we can apply use case driven principles [78] to undertake use case driven testing. That is, our UBT approach starts with use cases to derive functional system/integration test requirements (and/or high-level test design), and employs relevant UML models to drive the test process all the way through test planning, test analysis, test design, test generation and test evaluation.

Technically, use cases are related to the early stage of the SDLC, while test cases are associated with the latter stage of the SDLC. Use case driven testing leverages use cases with test case development and enables testing to be undertaken much earlier in the development process. This is a key MBT advantage (in Section 3.2.7) that use case driven testing aims to support. With use case driven testing, we can employ model-based tests to undertake SCT in two important aspects:

(1) Validating component specification with model-based tests in the form of test requirements and abstract test cases. This aids in uncovering errors/faults made in component specifications in the early phases of component development.

(2) Verifying component implementation with model-based tests in the form of test requirements and concrete test cases that are subsequently derived from the abstract test cases. This aids in detecting errors/faults made in component implementations in the latter phases of component development.

In this research, we incorporate use case driven testing in our MBSCT methodology for UML-based integration/system testing of software components.

3.3.4 General Approaches/Strategies for Applying UML Diagrams for Software Testing

According to the MBT principles (as described in Section 3.2), ordinary UML-based development models are usually not "test-ready" to be used directly for model-based testing. One important task of MBT with UML is to bridge the "test gaps" between ordinary UML-based development models and UML-based test models (in Section 3.2.6). There are two general approaches for applying UML diagrams to model-based testing:

1. Approach #1: Improving and augmenting the present UML diagram/model with particular annotations or testing-support artefacts for test enhancements to facilitate test derivation from the augmented UML model.

   Comparatively, Approach #1 has several advantages:

(1) Advantage #1: Using standard UML diagrams/models and UML-based specifications as the core basis for conducting test derivation and all MBT activities.

(2) Advantage #2: The augmented test information comes from two main sources:

   (a) Source #1: The augmented test information is mainly retrieved from UML-based specifications, such as UML use case templates and specifications (e.g. requirements, functions), sequence diagrams (e.g. sequentially time-ordered interactions),

class diagrams and specifications (e.g. class attributes, operations), data dictionary, etc.

(b) Source #2: If Source #1 is not sufficient, some additional commonly-used test enhancement information is used for testability improvement.

(3) Advantage #3: The annotated supplements are used mainly for UML-based test enhancements to facilitate test derivation, and do not change the normal process and activities of MBT with UML.

(4) Advantage #4: This approach is very straightforward and acceptable in MBT practice, and becomes a commonly-used approach in the MBT domain.

2. Approach #2: Transforming the UML diagram/model into some intermediate graphical representation, some intermediate notation or some other new model, or directly using these intermediate forms, from which test cases are derived.

In contrast, Approach #2 has some disadvantages:

(1) Disadvantage #1: The intermediate graphical representation may be domain specific or user-defined, and not known or familiar to all testers. This implies that using this approach may require a more demanding learning curve for the tester.

(2) Disadvantage #2: The introduced intermediate graphical representation or notation is often not as standardised as UML models. Consequently, the whole testing process contains two different sub-processes: while the front-end testing sub-process is UML-based, the subsequent backend testing process is not.

(3) Disadvantage #3: The introduced intermediate representation may complicate the testing process, and/or change the normal process or some activity of UML-based testing.

(4) Disadvantage #4: Both in practice and technically, it is very difficult to ensure that this transformation is correct, and/or to guarantee that both forms (the original UML diagram/model and the transformed intermediate representation) are equivalent in terms of test design, generation and evaluation.

Comparatively, the first approach has more advantages over the second. The first approach is also consistent with the integrated approach for bridging the identified “test gaps” (as described in Section 3.2.6.3 (3) above). Thus, our MBSCT methodology takes Approach #1 and is fully UML-based, and does not use any intermediate graphical representation. This research introduces a new testing-support mechanism of *test contracts* for testability improvement. Test contracts are represented with commonly-used assertions and applied as special test conditions/constraints on particular UML model elements/artefacts of testing interest. Test contracts and associated techniques are fully described from Chapter 4 onwards.

There is another issue to be considered and that is whether testing is applied directly to the software specification or to the software implementation. In general, UBT should follow the primary MBT concepts and principles (as described in Section 3.2). One of them is that UBT derives test cases from a UML-based model to test the implementation of the software specified by the relevant UML-based model. This is contrary to some views in the literature that focus on testing software designs in terms of UML design models, rather than testing the software implementation [5] [68] [114] [115]. In this research, we focus on testing the implementation of software components and systems by using test cases that are derived from UML-based models and specifications, and thus directly support dynamic testing as the ultimate testing goal. We take the viewpoint that software specifications/models are "verified (non-dynamic)" when their corresponding software implementations are tested (dynamic), as discussed earlier in Section 2.5.

3.4 Related Work

MBT/UBT has played a crucial role in the software testing domain. There are a number of MBT/UBT methods and techniques that have different paradigms, characteristics and perspectives. They rely on various testing-related aspects, such as relevant models for testing, available test artefacts, target test environments, test objectives, etc. A comprehensive study and review of current MBT/UBT in existing related research work could bring several benefits:

(a) Understanding relevant methodologies, and the strengths, benefits and limitations of existing MBT/UBT approaches;
(b) Helping the users of MBT/UBT to select suitable MBT/UBT approaches for the particular test requirements and test objectives;
(c) Aiding to identify new research issues, improve existing MBT/UBT techniques, and develop new MBT/UBT approaches.

This section comprehensively reviews research work related to MBT/UBT to provide a key foundation of the literature review, so that we can identify the important problems and limitations in MBT/UBT (to be described in Section 3.5). More details about the literature review of related work in MBT/UBT can be found in [177].

3.4.1 State-Based Testing

One of the classic MBT approaches is state model-based testing or state-based testing, which develops test cases by modeling the SUT as a state machine. Binder [24] presents how to develop testable state models and state-based test cases, including basic state machines, state mod-

els for object-oriented software (e.g. Mealy state machine, Moore state machine, Statecharts, etc), and state-based test design and testing strategies.

There are a number of research papers on state-based testing with UML state diagrams (or UML statecharts or state machines), such as [103] [105], [85], [72], [32] [34] [35] [36] [99] [4], [133], [82], [41], [129]. This section focuses on reviewing and discussing the first ten papers because they are most relevant to our research.

Offutt & Abdurazik [103] develop a testing technique that adapts state-based specification test data generation criteria to generate test cases (for implementation code) from UML state diagrams. A test data generation tool (UMLTest) is developed to automatically generate test data. This work is evaluated with an empirical study (i.e. based on a cruise control system). A limitation is that this work only uses a restricted form of UML state diagrams. This technique only considers enabled transitions while UML has five types of categorised transitions, which may result in some states not being entered or reached. This technique also is limited to class-level testing, and may not directly support integration testing. The authors claim that their work is the first formal testing technique that is based on UML models.

Kim et al. [85] discuss the application of UML state diagrams to class testing. According to their method, control flow is identified by transforming UML state diagrams into extended finite state machines (EFSMs), data flow is identified by transforming EFSMs into flow graphs, and then conventional data flow analysis techniques can be applied to test case generation based on data flow in UML state diagrams. The resulting set of test cases provides the capability of checking that classes are correctly implemented against the specifications written in UML state diagrams by testing whether class implementations establish the desired control and data flow as described in the specification. A limitation of this work is that it only focuses on unit testing of classes, and does not consider interrelationships between classes to support object-oriented integration testing using UML state diagrams.

Hartmann et al. [72] presents a UML-based integration testing approach by using UML statechart diagrams as the basis to generate black-box tests for unit and integration testing. With UML statechart diagrams, this approach models individual components by using a state machine to define the dynamic behaviour of each component, specifies component interactions, and annotates the state machines with test requirements to construct a global behavioural model of the composed statecharts. Then, test cases are automatically derived from the annotated statecharts and global behavioural model, and executed to verify component conformance behaviour. Their approach is evaluated with a simple example in a design-based test environment, the TnT environment (which consists of test generation tool TDE/UML and test execution tool TECS),

which integrates test generation and execution with commercial UML modeling tools, e.g. Rational Rose (which is now available as IBM Rational Software [76]). The work is claimed to support testing of middleware-based components (e.g. COM/DCOM and CORBA middleware).

Briand et al. have published a set of research papers on state-base testing with UML statechart diagrams [32] [34] [35] [36] [99] [4]. The work in [32] [35] proposes a methodology to automate the derivation of test cases from UML statechart diagrams for a given set of transition test sequences. The procedure required for automated derivation is based on a careful normalisation and analysis of operation contracts (event/action) and transition guard conditions written in the Object Constraint Language (OCL) [160]. The work in [34] investigates the cost-effectiveness of state-based testing for classes or class clusters modeled with UML state diagrams (i.e. flat statecharts), and focuses on the experimental evaluation of a well-known state-based testing strategy: round trip testing [24], with a series of three experiments in controlled experimental settings. Their results show that, in most cases, state-based testing techniques are not likely to be sufficient by themselves to detect most of the faults present in the code, and they need to be complemented with black-box functional testing, such as the category-partition method [110].

Test criteria for state-based testing are usually associated with states, transitions and/or predicates in state models. Offutt et al. [103] [105] present four useful test criteria at different levels of abstraction of state-based specifications, each of which requires a different extent of testing: (1) transition coverage criterion; (2) full predicate converge criterion; (3) transition-pair coverage criterion; (4) complete sequence criterion. These four test criteria are mainly used for class-level testing with state models.

3.4.2 Software Integration Testing with UML

This section reviews and discusses research work related to software integration testing with UML models [13] [14] [19] [20] [162] [72] [113] [122] [4]. We concentrate our attention on the first four papers because they are most relevant to our research.

Basanieri & Bertolino [13] presents an approach for UML-based integration testing, called Use-Interaction testing, which uses UML use case, class and sequence diagrams. They use UML use case diagrams to represent the system functionalities under test, UML class diagrams to define operations and attributes (at a high level of abstraction) required by classes for the interactions of their objects, and UML sequence diagrams to realise the functionality execution in the selected use case and to analyse the sequence of messages between components in the sequence diagram. They combine message sequence analysis and the category partition

method [110] to generate test data manually. Since this is preliminary work on UML-based integration testing as part of an ongoing research project, this approach can only generate some "abstract" test cases (which are not executable for dynamic testing).

In their subsequent work for the same research project, Bertolino et al. [20] extend their approach to develop a framework for testing component-based software with UML. They provide an overview of an integrated testing framework that will reuse and integrate several tools and techniques:

(a) UML Components [43] provides a modeling notation and a process for specifying component-based systems.

(b) Cow_Suite [14] is a UML-based test environment, originally developed in the area of object-oriented testing, which will be expanded to allow for the derivation of test cases from the UML specifications.

(c) Java-based CDT framework [19] supports component deployment testing.

Wu et al. [162] presents a new test model and relevant UML-based test adequacy criteria that can be used for UML-based integration testing of component-based software. The test model is used to define and analyse four key test elements, which can model the characteristics of the interactions among components and which must be considered during component-based testing:

(a) *Interfaces*: Integration and system testing need to test each component interface at least once as the interfaces activate components in the integrated environment.

(b) *Events*: Every event in the system regardless of its type (e.g. external/internal events) needs to be covered by some tests. Interfaces and events testing ensures that every possible interaction between components is exercised.

(c) *Context-dependence Relationships* model how interfaces and events interact, and reflect control sequences of objects in a component with respect to single interactions between actors and the component. Testing context-sensitive dependence relationships may serve to identify interoperability faults caused by improper interactions among different components in the integration context.

(d) *Content-dependence Relationships*: The interface dependence relationship can be derived from the function dependence relationship of the component interface, where one or more correlated functions, whose signatures are encapsulated with the interface, may be executed to perform the requested service. Content-sensitive dependence relationships can provide valuable additional information in generating test cases and detecting faults.

Wu et al. also use UML behavioural diagrams to describe relevant component relationships with an illustrative example of an ATM sever component, where interaction (i.e. collabo-

ration/sequence) diagrams and statecharts are used to specify context-dependence relationships, and collaboration diagrams and statecharts are used to specify content-dependence relationships.

This paper also lists some UML-based test adequacy criteria to test key model artefacts for context/content-dependence relationships in their test model: each transition, each sequence, each context-dependence relationship and content-dependence relationship in the relevant UML diagrams have to be tested at least once.

The main research results presented in this paper are the definition of the four key test elements and relevant discussions with UML diagram illustrations, as well as some UML-based test adequacy criteria listed (but without any further discussions). While these test elements and test criteria are useful to test component-based software, this work is at the stage of "approach development and description". This paper does not currently discuss and give practical ways on how to use their approach to generate actual test cases and oracles for component-based testing. Their so-called "test model" mainly illustrates the context/content-dependence relationships defined by the authors, and constructing a real test model abiding by important MBT principles (as described earlier in Section 3.2.6) requires additional work to effectively drive test generation from the test model.

In addition, the authors made several assumptions in their work, including: (i) assuming that each individual component has been adequately tested by the component providers when testing component-based software; (ii) assuming that each interface only includes one operation, and the references to the interfaces and to the operation are identical. These assumptions imply that their work currently considers only some simplified situations, which could have limitations in applying their approach to actual component-based testing practice.

Hartmann et al. [72] carry out software integration testing using UML statechart diagrams (see our review of this work as described in Section 3.4.1 above).

3.4.3 Software System Testing with UML

This section reviews and discusses research work related to software system testing with UML models [30] [73] [102] [9] [132].

Briand & Labiche [30] present a system test methodology that derives functional system test requirements from UML-based analysis models that are produced at the end of the analysis development stage. These UML analysis artefacts are represented with use cases (use case diagrams and descriptions), interaction (sequence or collaboration) diagrams (associated with each use case), class diagrams (composed of application domain classes), a data dictionary (describing each class, method and attribute) (which is also an *assumption* of their approach). First, they build one activity diagram per actor in the system to model and capture sequential dependencies

and constraints between the use cases related to the actors, and generate legal sequences of use cases according to the sequential dependencies specified in the activity diagram. These use case sequences and dependencies constitute the first component of system test requirements. Then, they use the system sequence diagrams augmented with precise OCL guard conditions (which is an *assumption* of their approach) to describe the associated use cases, derive operation sequences of use case scenarios to be executed and tested, and identify test oracles for each operation sequence based on the OCL-specified postconditions of operations in the sequence. Finally, for a given use case, they formalise all identified operation sequences to be tested, related initial conditions and test oracles into a decision table, which is used as a formal set of system test requirements. This paper uses a library system as an example.

However, this work does not currently address how to generate actual system-level test cases and oracles by using the test requirements derived with their methodology. Producing the complete test requirements for the SUT requires additional work, and the methodology needs to be improved and evaluated with more complex case studies. In addition, this approach relies on a number of assumptions they make regarding the way a UML analysis model is developed, such assumptions as the data dictionary, and OCL constraints on the UML analysis artefacts under test (e.g. OCL expressions for invariants of each class, and for pre/post-conditions of each operation). Such assumptions implicitly presume that an ordinary UML analysis model would already have certain "good" testability properties, and be "test-ready" to be used directly as a target test model for test derivation, or that it would be "easy" to fulfil the relevant requirements for the assumptions, making these assumptions virtually acceptable or valid in UBT practice. However, these "virtually-acceptable assumptions" are not realistic and are linked to certain "*test gaps*" that are required to be bridged in UBT practice, which is regarded as a very important part of MBT principles as described earlier in Section 3.2.6. Among many other testing issues, how to improve testability and obtain testability requirements are among the most crucial and difficult testing tasks. A key focus of our research is on model-based testability improvement for effective test model construction in our MBSCT methodology.

Hartmann et al. [73] describes a tool-support approach for automatically generating and executing system tests. They use UML use case diagrams and activity diagrams to model the dynamic behaviour of systems, manually annotate the behavioural models (activity diagrams) with test requirements, and refine the activity diagrams in more detail for the test generator/executor. System tests are automatically generated from the behavioural models, and then executed with their TnT environment. The TnT environment is developed originally to support unit and integration testing with UML statecharts [72], and currently to support system testing with UML use case diagrams and activity diagrams [73].

Nebut et al. [102] [101] carry out software system testing using UML use cases and scenarios (which is further reviewed in Section 3.4.4 below).

3.4.4 Software Testing with UML Use Cases and Scenarios

UML use case diagrams and scenarios mainly focus on describing system requirements and analysis to construct use case models and analysis models, which capture system requirements and (integration) functionalities at high levels of abstraction. Accordingly, UML use case diagrams and scenarios are mainly used as a basis to derive system/integration test requirements, which can further drive design and generation of system/integration tests for system/integration testing. UML use case diagrams and scenarios (use case view) are often used in conjunction with UML sequence diagrams (behavioural view) to undertake system/integration testing. This section reviews and discusses research work related to software testing based on UML use case diagrams and scenarios.

Nebut et al. [102] present a use case driven approach for automatic system test generation. They enhance UML use cases with requirement-level contracts based on use case pre and post conditions to describe system requirements [101], and use UML sequence diagrams to describe use case scenarios. They propose a two-phase method to automatically generate functional test scenarios from requirement artefacts: (i) the first phase aims at generating test objectives (i.e. relevant extracted paths of the valid sequences of use cases) from a use case view of the system (with use cases, contracts and coverage criteria); (ii) the second phase aims at generating test scenarios from these test objectives (by transforming the test objectives into the test scenarios using sequence diagrams). Then, they synthesise test cases from the derived test objectives and test scenarios with their tool-support transition system. The approach is empirically evaluated with three case studies in terms of statement coverage with the generated tests.

Basanieri & Bertolino [13] apply UML use case diagrams and other UML diagrams (class and sequence diagrams) to UML-based integration testing (see our review of this work as described in Section 3.4.2).

Briand & Labiche [30] apply UML use case diagrams, in conjunction with UML class and collaboration/sequence diagrams, to produce UML-based analysis models and derive system test requirements for UML-based system testing (see the review of this work as described in Section 3.4.3).

Hartmann et al. [73] apply UML use case diagrams and activity diagrams to UML-based

system testing (see the review of this work as described in Section 3.4.3).

Tsai et al. have published several research papers related to scenario-based testing, such as [142] [143] [144] [145] [146] [147]. The work in [142] proposes a scenario-based functional regression testing method, which is based on end-to-end integration test scenarios and focuses on the functional correctness of integrated systems from the end user's viewpoint. They represent test scenarios in a template model that embodies both test dependency and traceability information, and use a ripple effect analysis to identify all affected, including directly or indirectly, scenarios, and thus the set of test cases selected for regression testing. The work in [143] presents a process to develop adaptive object-oriented scenario-based test frameworks for testing embedded systems. Their process uses techniques such as design-for-change, design patterns, scenarios, ripple effect analysis and regression testing, and is illustrated with a test example of a mobile phone system. The work in [144] presents a scenario-based object-oriented framework to test distributed systems rapidly and adaptively by using several techniques, such as scenario modeling, state modeling, design patterns, verification patterns, regression testing, ripple effect analysis and test simulation. Their framework is illustrated in testing a supply chain management application developed using web services. The work in [145] proposes a scenario-based web service testing framework that provides three main distributed components: (a) test masters that manage scenarios and generates test scripts; (b) test agents that dynamically bind and invoke web services; (c) test monitors that capture synchronous /asynchronous messages sent and received, attach timestamp, and trace state change information. As they claim, the benefit of using this framework is that the user only needs to specify system scenarios based on the system requirements without needing to write test code. Then, the framework generates test scripts, executes tests, collects and monitors test results, and evaluates test results at runtime. The work in [146] presents a scenario-based object-oriented framework for adaptive and rapid testing, which improves their earlier work in [143] and includes new features such as database support, regression testing, assurance-based testing, three-tier architecture and web-based tool support. Their framework takes test scenario specification as input, prepares data for test execution, performs test execution, and evaluates test results with a database support.

3.4.5 Software Testing with UML Sequence Diagrams

This section reviews and discusses research work related to software testing based on UML sequence diagrams [13] [30] [102] [131] [89] [128] [54] [10] [11] [135] [130] [61] [80].

Basanieri & Bertolino [13] use UML sequence diagrams to realise functionality execution in the selected use case and analyse the sequence of messages between components in the sequence diagram for UML-based integration testing (see our review of this work as described in

Section 3.4.2). Briand & Labiche [30] use the system sequence diagrams to describe the associated use cases, derive operation sequences of use case scenarios to be executed and tested for UML-based system testing (see the review of this work as described in Section 3.4.3). Nebut et al. [102] use system-level sequence diagrams in the second phase of their use case driven approach to system testing (see the review of this work as described in Section 3.4.4). They employ sequence diagrams to represent the instantiated use case scenarios, which are used to replace the instantiated use cases in the test objectives derived in the first phase, and obtain the sequences of scenarios. They then transform the scenario sequences into test scenarios using strong sequential composition.

Many reported methods require transforming UML diagrams into some intermediate graphical representations. For example, the work in [131] presents a method of generating test cases from UML sequence diagrams. This method transforms a UML sequence diagram into an intermediate graph called a sequence diagram graph that augments each node in the converted graph with test information (retrieved from use case templates, class diagrams and data dictionary), and then the augmented converted graph is traversed to generate test cases (e.g. test input, output, postcondition). The work in [89] presents a test case generation method based on UML sequence diagrams augmented with OCL expressions. This method firstly converts a sequence diagram into an intermediate scenario tree representation, and then selects and transforms conditional predicates (pre/post-conditions in OCL) to generate test data. The work in [128] describes a testing method to generate test cases from UML sequence diagrams using dynamic slicing. This method identifies message guards on sequence diagrams, creates dynamic slices with respect to the slice, and generates the test case with respect to the slice. The generated test cases satisfy a slicing coverage criterion formulated as a test adequacy criterion. However, the above methods are based on using some intermediate graphical representation that is not as standardised as the UML notation. Accordingly, this may result in the relevant testing method that is not suitable for a tester on the component user side who is unfamiliar with these intermediate representations in UBT practice.

On the other hand, some reported methods require directly using some non-UML standard notations or graphical representations to base their methods on, which have similar deficiencies as the above. For example, Dinh-Trong et al. [54] introduce a graphical representation called the variable assignment graph (VAG) that combines relevant information from UML sequence diagrams and class diagrams for UML design models. A symbolic execution based approach is used to derive test input constraints from the paths of this directed graph. The derived test inputs help fault detection in UML design models. Following this work, Bandyopadhyay & Ghosh [10] [11] develop an extended variable assignment graph (EVAG) that integrates rele-

vant information from UML sequence diagrams and state diagrams to generate test inputs. Their own mutation analysis on their pilot study shows that the EVAG is more effective than the VAG by using more precise information describing the effects of message sequences in the state machine models.

A key feature of a sequence diagram is its feature of sequential ordering in arranging interaction messages, which can drive and facilitate developing corresponding test sequences in test case generation. In [135], a method is presented to generate test sequences from a combination of UML sequence diagrams, which is complemented by the use of state diagrams. A sequence diagram describes a set of test sequences differing in terms of the different states of the participating objects. An attached state diagram for each participating object describes its states (and the transition behaviour between these states). Each combination of initial state configuration and initialisation sequences describe at least one test case in the set. In [130], a method is presented to generate test sequences from UML 2.0 sequence diagrams. The method first transforms a sequence diagram into an intermediate form called a sequence dependency graph from which test sequences are generated. They define four types of message dependency relationships (indirect message dependency, direct message dependency, simple indirect message dependency, and simple direct message dependency), and derive message sequences using the "execution occurrence" feature of sequence diagrams. The derived message sequences are incorporated into the transformed sequence dependency graph. Finally, a traversal algorithm is described to generate test sequences from the sequence dependency graph. However, in the above methods, except for the test sequences being generated, it is not very clear how individual tests and the necessary checking in a test case are generated and arranged in the specific derived test sequence.

UML sequence diagrams can be used as a basis to drive the development of model-based testing tools. Fraikin and Leonhardt [61] develop a test tool SeDiTeC that supports automated generation of test stubs based on their refined sequence diagrams, which are complemented by test case data sets consisting of parameters and return values of method calls. For classes and their methods whose behaviour is specified in sequence diagrams and given the corresponding test case data sets, the test tool can automatically generate test stubs thus enabling early testing before the completion of the implementation phase. Validation compares the test case data sets with the observed data, and also includes validation of class method pre/post-conditions. However, this test tool relies on a particular platform, and is developed with the UML CASE tool Together Control Centre (which is now available as Borland Together [29]).

Javed et al. [80] develop a model transformation prototype tool to support their model-driven testing method, which utilises the model transformation technology of model-driven ar-

chitecture [106] to generate unit test cases from a platform-independent model (PIM) of the SUT. Their method is based on UML sequence diagrams (forming the platform-independent model), where the generation of unit test cases consists of two steps. The first step models a sequence diagram as a sequence of method calls, which is automatically transformed into a general unit test case model xUnit [163] by applying model-to-model horizontal transformation (PIM to PIM transformation) using Tefkat [140]. The second step generates concrete and executable JUnit [81] test cases from the xUnit model by applying model-to-text vertical transformation (PIM to platform-specific model transformation) using MOFScript [95]. While this method takes advantage of MDA technology for automatic test case generation from UML sequence diagrams, it is not very clear that how the initial sequence diagrams, which they use for the two-step model transformations, would contain adequate testable information for generating unit test cases by using their prototype tool. As indicated earlier in Section 3.2.6, bridging the actual "test gaps" between sequence diagrams and ready-to-use test models is a challenging task in UBT practice.

Test criteria based on UML sequence diagrams are usually associated with messages, a sequence of messages, and/or start-to-end message paths in sequence diagram models. Among other test criteria, the all-paths criterion requires each possible message path to be taken and exercised at least once. This is an important requirement of adequate testing of sequences of messages or equivalent representations. Several papers have used a similar all-paths criterion in their testing techniques, with coverage of message sequences in sequence diagrams [13] [61] [131] [89] [83], message sequences in collaboration diagrams [1] [162], or use case sequences (i.e. all scenarios) in use case diagrams [30]. However, many papers do not address a critical problem associated with the practicality of the all-paths criterion: there is a possibility to have an infinite number of all start-to-end message paths in sequence diagrams. To deal with the problem of such infinity, Binder suggests using a subset of all start-to-end message paths in a sequence diagram [24]. This indicates that the all-paths criterion would not be always feasible in testing practice, and the problem of its practicality remains to be resolved.

The use of UML sequence diagrams for integration testing was evaluated in several studies. For example, Abdurazik et al. [2] [3] [83] present a single project experiment on the fault revealing capabilities of model-based test sets generated from UML statecharts and sequence diagrams. Their results show that the sequence diagram tests have better capability of revealing integration level faults than the statechart tests, and they recommend that sequence diagrams should be used for integration level testing.

3.4.6 Other Related Work

This section briefly reviews other MBT-related work (e.g. non-UML based) that is in the literature but is not directly related to the scope of this research, in order to show the acknowledgment of other MBT-related work with our comprehensive literature review.

One worth-mentioning type of MBT (non-UML based) is formal testing that employs model-based formal specifications for software testing [182] [189] [190]. The work on this type of MBT applies relevant formal specifications, for example, finite state machines [192] [183] [186] [191], labelled transition systems and input output transition systems [198] [141] [199], Z specifications [187] [188], etc. Because formal testing relies on mathematically based formal specifications (e.g. mathematically based languages, notations, syntax and/or semantics) to specify software models for MBT, they are used mostly within theoretical research academia, but are not favoured by software engineers/testers in industry. Accordingly, this type of MBT is much less popular and practically useful than UBT. It is outside the scope of this research.

Based on the OMG UML 2.0 standard, the UML Testing Profile (UTP/U2TP) is a domain-independent test modeling language for test case specification, test data specification, test deployment, and test result visualisation [193]. However, since its perception (e.g. the OMG UTP 1.0 release in 2005 [194]), the UTP does not have high research interests in academia, and is not widely used in software engineering industry as expected [196] [184] [185] [201] [181] [195] [197] [200]. A straightforward reason would be that ones in both academic and industry communities have already employed UML models (with UML diagrams) as test models well before the UTP perception. The more important reasons are that the current UTP has a number of itself own limitations [197] [200]. For example, (1) the UTP does not directly support test case generation except for test specification; (2) the UTP is not a testing methodology that can provide effective testing processes, techniques and guidelines to derive actual test cases and to undertake fault detection and diagnosis; (3) test specification documents described with the UTP have inadequate readability just as the UTP has certain inconsistencies to the UML 2.0 standard; (4) there are no effective and sufficient testing tools supporting software testing using the UTP in MBT practice.

Accordingly, this research does not use the UTP for test specification. The main objectives of this research are to address more important challenging research problems (as outlined in Section 1.1 and as further reviewed in Section 3.5 below) beyond the way of test modeling and specification as with the UTP. The new MBSCT methodology introduced by this research is a novel comprehensive SCT methodology for test model construction, component test case design and generation, component testability improvement, component fault detection and diagnosis, and component test evaluation (as outlined in Chapter 1 and as further described in more detail from Chapter 4 onwards).

3.5 Analysis and Discussion

Based on our literature review, we have identified a number of important problems and limitations that remain unresolved in MBT/UBT, which forms a very important part of the research results of our literature review. These issues particularly appear in many representative MBT/UBT approaches reported in the literature (that have been highly cited by a larger number of research papers), such as [103] [72] [30] [162] [105] [102].

The following analyses and discusses these important problems and limitations in MBT/UBT:

1. MBT/UBT practice and the entire software development process

The current MBT/UBT practice (including approaches, activities) has not been fully integrated (at least not in an effective and efficient way from our literature review) into the entire software development process, and/or the project/organisation's continuous test process. This MBT/UBT limitation reflects on three main aspects as follows:

(a) On the one hand, because the use of MBT approaches means a significant paradigmatic change from IBT or other traditional testing approaches, there are some obstacles in technology transfer of MBT into testing organisations, so the overall process of software development and testing must be adapted.

(b) On the other hand, because of the aforementioned problem, software models used for test generation are not incorporated appropriately with software artefacts produced from the software development process, or software models are defined merely by and for a specific MBT approach in use. This can cause the use of an MBT approach not to be cost-effective.

(c) More important, there is no unified software development process that integrates MBT/UBT activities effectively and efficiently into the entire SDLC.

This problem in MBT/UBT was also observed in several prior studies [46] [25] [21] [47] [48] [26]. Their studies focused on relevant process and organisation issues and impacts of MBT in terms of research findings, project experiences, learned lessons, encountered challenges, constructive recommendations and/or future trends.

This issue is very important and should be resolved. However, many of the abovementioned representative MBT/UBT approaches (as listed above in the first paragraph of this section) do not well address this problem. In this research, we argue that the integration of MBT and MBD activities should allow both to work collaboratively together in the SDLC process to deal with this crucial problem in MBT/UBT. Our proposed MBT definition (in Section 3.2.5)

has provided a conceptual basis to address this problem. This research aims to overcome this limitation in MBT/UBT practice by using the *Model-Based Integrated SCT Process* developed for the new MBSCT methodology (as described earlier in Section 1.1 and Section 1.2).

2. "Test Gaps" and inadequate model-based testability in MBT/UBT

As reviewed in Section 3.2.6.3, there exist certain "test gaps" causing inadequate model-based testability in MBT practice. However, this issue has not been recognised appropriately in the literature and there are some misunderstandings about how to create a test model. One misunderstanding is a belief that to build a test model exclusively for MBT from scratch can be done without using existing software development models. Another misunderstanding is a belief that it is possible for a tester to "fully reuse" a simple or ordinary software model directly from software development as a test model with no modifications. These misunderstandings can result in the waste of software development resources or inadequate model-based testability. Accordingly, a lack of recognition of the "test gaps" issue is a practical obstacle to undertake MBT/UBT effectively.

Our literature review shows that most MBT/UBT approaches reported do not address this problem, including the representative MBT/UBT approaches as mentioned above in the first paragraph of this section. The "test gaps" issue is very important and should be resolved. In this research, we argue that it is important to address the correlation between "test gaps" and inadequate model-based testability. This research aims to overcome this MBT/UBT limitation by bridging the "test gaps" and improving model-based testability with the *Test by Contract* technique and the *Test-Centric Remodeling* strategy developed with the new MBSCT methodology (as described earlier in Section 1.1 and Section 1.2).

3. Comprehensiveness in MBT/UBT approaches

Dias Neto et al. conduct a comprehensive survey of MBT/UBT approaches [47], and indicate that: "MBT approaches usually cannot represent and test non-functional requirements, such as software useability, security, performance, and reliability" [48]. This corresponds with the findings of our literature review. In addition to this type of deficiency, our literature review shows that, due to various reasons, most MBT/UBT approaches reported in the literature lack certain methodological comprehensiveness: they usually contain only limited (usually one or two) individual testing techniques, causing them not to have sufficient core testing features and capabilities, such as testability improvement, fault detection, diagnosis and localisation. For example, almost all of the abovementioned representative MBT/UBT approaches do not adequately cover these core testing features and capabilities. Furthermore, some of these approaches do not show important detailed and operational descriptions on how to apply them to generate actual test cases and oracles, such as the UBT approaches by [30] (as reviewed in Sec-

tion 3.4.3) and [162] (as reviewed in Section 3.4.2).

This research aims to address this limitation in MBT/UBT practice. The new MBSCT methodology is developed to be a comprehensive UML-based SCT methodology that has a number of effective testing techniques and processes, core testing features and capabilities (as described in Section 1.1 and Section 1.2).

4. Validation and evaluation of MBT/UBT approaches

In their review of MBT/UBT approaches [47], Dias Neto et al. also indicate that: "most MBT approaches are not evaluated empirically, and/or not readily transferred from the academic research environment to the practical industry environment" [48]. This corresponds well with the research results of our literature review. In particular, our literature review shows that most MBT/UBT approaches reported in the literature are presented only with limited individual testing examples, which means that most MBT/UBT approaches have not been comprehensively validated and evaluated. This is also seen in the abovementioned representative MBT/UBT approaches.

This research aims to address this MBT/UBT limitation. Our MBSCT methodology has been validated and evaluated comprehensively with two full case studies, in terms of the most important SCT/MBT activities, such as test model construction, component test design and generation, component testability improvement, component fault detection and diagnosis (as described in Section 1.1 and Section 1.2).

In addition, there are some other MBT/UBT problems identified in the literature review [177]. For example, many MBT/UBT approaches reported in the literature appear to be manual testing and/or are deficient in tool-support test automation, or they usually do not have adequate test criteria to enhance and ensure testing quality, and so on. These limitations are also seen in many of the abovementioned representative MBT/UBT approaches. In the scope of this research, we particularly focus on addressing the above-stated four most important problems and limitations in MBT/UBT with the development of the new MBSCT methodology.

3.6 Summary

This chapter has provided a comprehensive literature review of the important concepts, principles, characteristics, methods and techniques of MBT in general and UBT in particular. We discussed a number of important MBT/UBT issues and associated aspects (as raised earlier in Section 3.1). Both MBT and MBD must be integrated and collaboratively work together as part of the SDLC. UBT advances MBT towards a mainstream MBT approach. In this research, we employ MBT as the primary software testing approach, UML as the object-oriented software mod-

eling language, and UBT as the major type of MBT approach.

The comprehensive literature review and further research work undertaken in this chapter have attained a number of research results and findings (including new concepts and definitions, important problems and limitations in MBT/UBT), which make original contributions to the body of knowledge in the main research areas of MBT and UBT in the literature. The following summarises our main research results in this chapter and describes how we intend to apply them in this research:

(Note that the main problems and limitations identified in MBT/UBT are described in Section 3.5 above.)

1. A study of model-based tests (in Section 3.2.4)

As described in Section 3.2.4, the initial form of model-based tests derived directly from test models is "abstract", and such abstract test cases are not ready to be used for dynamic testing. Accordingly, it is very necessary to have some practical testing technique that can support transforming abstract test cases into concrete test cases suitable for test execution for dynamic testing. For this purpose, we develop a test mapping technique for SCT, *Component Test Mapping*, which is an integral part of our MBSCT methodology. This technique will be introduced in Chapter 4, and further discussed and applied in Chapter 8.

2. A new definition of model-based testing (in Section 3.2.5)

We have proposed a new definition of model-based testing, which covers the main MBT activities, tasks and goals. A key characteristic of the new MBT definition is that it clearly emphasises the integration of MBT into the software development processes, enabling MBT and MBD to work together in the SDLC. Our proposed MBT definition aims to use an integrated testing process to overcome the first outstanding limitation in MBT/UBT (as described in Section 3.5). We incorporate this important feature of the new MBT definition in the development of the *Model-Based Integrated SCT Process* with the new MBSCT methodology, which will be introduced in Chapter 4 and applied in Chapter 5 onwards.

3. A new test model definition (in Section 3.2.6)

We have proposed a new test model definition, and emphasised the relevant implications and importance of test models in MBT practice. A test model for MBT is not exactly the same as its associated development model from MBD. A good test model should be reasonably simple and/or more abstract than the concrete implementation of the SUT, but it also must be adequately precise for testing requirements. We apply these useful guidelines to develop test models with our MBSCT methodology.

More important, we must bridge the identified "test gaps" and improve model-based testability for effective test model construction, in order to resolve the second outstanding limitation in MBT/UBT (as described in Section 3.5). This is an important starting point that has motivated this research to develop the *Test-Centric Remodeling* strategy and the *Test by Contract* technique with the new MBSCT methodology, which will be described in Chapter 4 to Chapter 7.

4. A new definition of UML-based testing (in Section 3.3.1)

A new UBT definition is proposed based on our MBT definition. UBT is the major type of MBT approach we use in this research to obtain the benefits of the standardised notations and rigorous semantics of the UML. Based on this UBT definition, our MBSCT methodology is a new approach for UML-based testing of software components and systems (i.e. UML-based SCT).

5. A core UML subset for SCT (in Section 3.3.2)

Among 13 types of UML diagrams, we have selected five main UML diagrams to form our core UML subset, UML–SCT. Our selection was justified based on a major point: our literature review shows that UML diagrams in UML–SCT are most commonly used for object-oriented software behaviour modeling and testing (as reviewed in Section 3.3.2, Section 3.4.1 to Section 3.4.5). Our MBSCT methodology is a UML-based SCT approach to develop UML-based test models and UML-based component tests that are described by UML diagrams in UML–SCT.

6. A study and review of use case driven testing and scenario-based testing (in Sections 3.3.2 to 3.3.3, and Sections 3.4.2 to 3.4.5)

We have studied the main concepts and reviewed related work of use case driven testing and scenario-based testing. Motivated by this review, we introduce the *Scenario-Based Component Integration Testing* technique in our MBSCT methodology. This technique employs scenarios as concrete use case instances to capture behavioural interaction dynamics in the integration context, which aims to develop test scenarios and associated test sequences for UML-based integration testing of software components. This technique will be introduced in Chapter 4 and applied from Chapter 5 onwards.

By providing a comprehensive literature review and further research results in MBT/UBT, the research work in this chapter has created an advanced methodology foundation to support the development of the new MBSCT methodology, which will be described from Chapter 4 onwards.

Chapter 4
Model-Based Software Component Testing: A Methodology Overview

4.1 Introduction

The early chapters (Chapter 2 and Chapter 3) of this thesis presented a comprehensive literature review and associated further research results, which creates a solid conceptual and methodological foundation to support the development of the new methodology. Chapter 4 to Chapter 9 of this thesis describe the main body of the new methodology, which is the core part of this research. Starting from Chapter 4 onwards, we formally introduce and describe the new methodology of *Model-Based Software Component Testing* (MBSCT) we have developed in this research [167] [169] [171] [172] [173] [174] [175] [176] [179].

This chapter presents an overview of the MBSCT methodology. First, Section 4.2 gives an overall methodology summary. Then, we describe the major MBSCT methodological components (in Section 4.3), the MBSCT framework for UML-based SCT (in Section 4.4), the main MBSCT methodological features (in Section 4.5), and the core MBSCT testing capabilities (in Section 4.6). Finally, Section 4.7 presents a summary of this chapter. More technical details and application of the MBSCT methodology are further discussed and illustrated with case studies in the subsequent chapters of this thesis.

4.2 Methodology Summary

The MBSCT methodology is a novel hybrid UML-based SCT approach that aims to address a number of the most important challenging research problems in SCT/MBT (as identified earlier in Section 1.1 and as further reviewed in Section 3.5), and to achieve the following goals:

(1) Integrating UML-based testing into a unified UML-based software process as part of the SDLC, enabling the utilisation of a consistent UML-based approach and specification for all UML-based component development and testing activities;

(2) Bridging the "*test gaps*" between ordinary UML-based software models (which are not test-ready or non-testable) and target test models (which are test-ready or testable) to support UML-based SCT;

(3) Improving model-based component testability for effective test model construction, com-

ponent test design and generation, component fault detection, diagnosis and localisation;

(4) Focusing testing priority on the development of scenario-based test models and scenario-based test cases to support component integration testing that bridges component unit testing and component system testing;

(5) Deriving model-based component test artefacts to generate target component test cases (e.g. CTS test case specifications);

(6) Incorporating SCT activities with widely-used proven software concepts and practices (e.g. object orientation, UML modeling, Unified Process, use case driven principles [78] to develop a comprehensive integrated SCT approach with unique methodological features and testing capabilities;

(7) Undertaking comprehensive methodology validation and evaluation to achieve the required level of component correctness and quality;

(8) Advancing SCT/MBT approaches and practices to produce trusted quality software components with better effectiveness and efficiency in CBSE.

The MBSCT Methodology is developed to possess five major methodological components (process/technique/strategy), a three-phase testing framework, six main methodological features and six core testing capabilities, which altogether represent four modules of the MBSCT methodology. Figure 4.1 illustrates the composition (static) of the four MBSCT modules. Table 4.1 shows an overall outline of the MBSCT methodology. The subsequent Sections 4.3 to 4.6 further describe each MBSCT module and their methodological collaboration (dynamic) to support the entire MBSCT methodology.

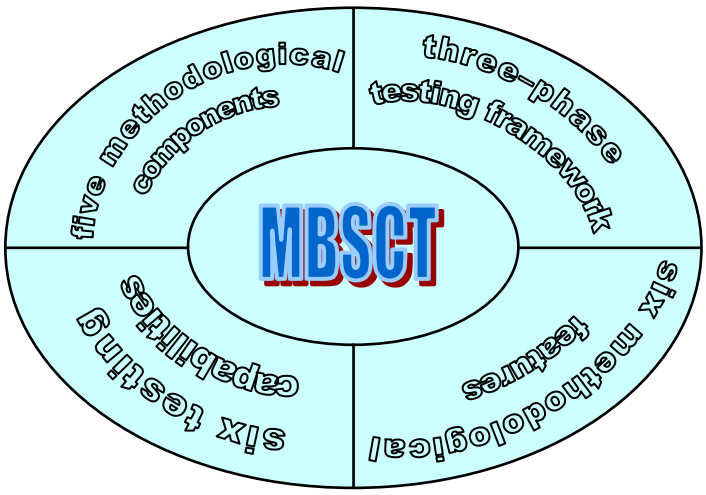


Figure 4.1 The MBSCT Methodology: Four Composite Modules

Table 4.1 The MBSCT Methodology: an Overall Outline

Five Major Methodological Components	1. Model-Based Integrated SCT process
	2. Scenario-Based Component Integration Testing technique
	3. Test by Contract technique
	4. Test-Centric Remodeling strategy
	5. Component Test Mapping technique
Three-Phase Testing Framework	1. Test model construction
	2. Component test design and generation
	3. Component test evaluation
Six Main Methodological Features	1. Model-based
	2. Process-based
	3. Scenario-based
	4. Contract-based
	5. FDD-based
	6. Mapping-based
Six Core Testing Capabilities	1. Test model construction
	2. Component test design and generation
	3. Component fault detection, diagnosis and localisation
	4. Adequate test artefact coverage
	5. Component testability improvement
	6. Adequate component fault coverage and diagnostic solutions

4.3 Major Methodological Components

This section introduces the five major methodological components developed with the MBSCT methodology, including the Model-Based Integrated SCT process (in Section 4.3.1), Scenario-Based Component Integration Testing technique (in Section 4.3.2), Test by Contract technique (in Section 4.3.3), Test-Centric Remodeling strategy (in Section 4.3.4), and Component Test Mapping technique (in Section 4.3.5).

4.3.1 Model-Based Integrated SCT Process

The *Model-Based Integrated SCT* process is the most important methodological component and provides an overall process model for the entire MBSCT methodology. Its primary objective is to support the MBSCT methodology's three goals (1), (2) and (6) (as described in Section 4.2):

(a) Supporting goal (1) by addressing the first challenging research problem (as described

earlier in Section 1.2 and Section 3.5) and incorporating the new MBT/UBT definition (i.e. Definition 3–1 in Section 3.2.5 and Definition 3–3 in Section 3.3.1), enabling the MBSCT methodology to have model-based and process-based features;

(b) Supporting goal (2) by providing a key process foundation for bridging the "test gaps" for effective UML-based SCT;

(c) Supporting goal (6) by integrating this SCT process with commonly-used proven software concepts and practices (e.g. object-oriented methods, UML modeling, Unified Process, use case driven principles).

This SCT process integrates software component development and testing activities into a unified UML-based software process as part of the core phases of the SDLC, and enables the use of a consistent UML-based approach and specification for the systematic development of UML-based test models, model-based component tests, and model-based test evaluation. It is developed to be an extension of the general iterative and incremental development approaches /processes (e.g. Unified Process) to the new domain of UML-based SCT. The entire SCT process consists of two main parallel workflow steams: *model-based SCD* (MBSCD) and *model-based SCT* (MBSCT), which work collaboratively together. Figure 4.2 illustrates both workflows and their relationships, where the left-side half of the figure represents the MBSCD process and the right-side half of the figure represents the MBSCT process. Both workflows follow the proven iterative and incremental approach of the UP principles, and jointly apply UML modeling to component development and testing.

In particular, the integrated SCT process guides the iterative and incremental construction of UML-based test models for SCT in combination with parallel UML-based development models for SCD, which is described as follows:

(1) *MBSCD Process*

In the context of object-oriented SCD, the MBSCD process is composed of a number of UML-based development steps (marked D0, D1, D2, …), which feature Object-Oriented Analysis/Design/Programming (OOA/OOD/OOP) and produce a set of UML-based software models for SCD (called *SCD models*) at different modeling levels, including the Use Case Model, Object Analysis Model, Object Design Model, and Object Implementation Model.

The *central control point* "ID" evaluates and manages the iterative/incremental process of development/modeling. In particular, when the current development/modeling step is not completed, or when the current development/modeling step is completed but the entire development/modeling has NOT been completed, the "ID" point evaluates the current development/modeling step and decides which next iterative/incremental "D" step (among Step D1 to Step D4) is to be carried out.

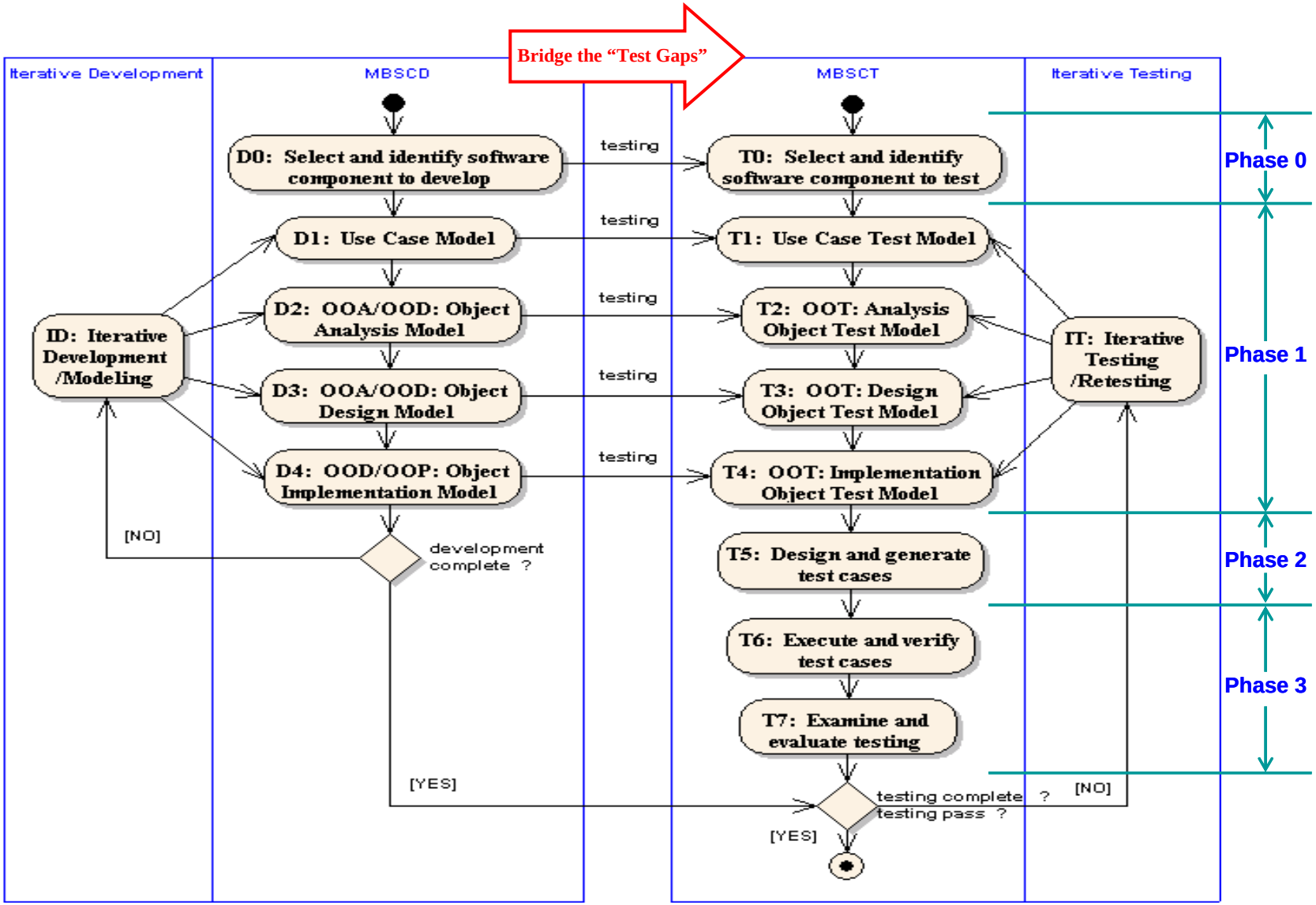


Figure 4.2 MBSCT Methodology: Model-Based Integrated SCT Process

(2) *MBSCT Process*

In parallel, as shown on the right-side half of Figure 4.2, the MBSCT process is linked and works closely with the MBSCD process. Working in the counterpart context of object-oriented SCT (object-oriented testing or OOT), the MBSCT process covers a number of corresponding UML-based testing steps (marked T0, T1, T2, …), which build a group of UML-based test models for SCT (called *SCT models*) at different test levels, including the Use Case Test Model, Analysis Object Test Model, Design Object Test Model, and Implementation Object Test Model.

The *central control point* "IT" evaluates and manages the iterative/incremental process of testing/retesting. In particular, when the current testing/retesting step is not passed, or when the current testing/retesting step is passed but the entire testing/retesting has NOT been completed, the "IT" point evaluates the current testing/retesting step and decides which next iterative/incremental "T" step (among Step T1 to Step T4) is to be carried out.

The integrated SCT process guides what types of test models need to be developed in terms of the different levels of use case and object test models. This entails the iterative and incremental development of a series of UML-based test models in different test steps (e.g. MBSCD/MBSCT Steps D1/T1 to D4/T4), as described in (1) and (2) above. Note that a "T" (Test) step in the MBSCT process corresponds to a parallel "D" (Development) step in the MBSCD process at the same modeling level. Usually, a later "D" step is mainly based on its preceding "D" step, for example, the Object Design Model in MBSCD Step D3 is constructed based on the Object Analysis Model in its preceding MBSCD Step D2. However, a later "T" step is mainly based on its parallel "D" step and its preceding "T" step, for example, we construct the Design Object Test Model in MBSCT Step T3 based on the Object Design Model in its parallel MBSCD Step D3 and the Analysis Object Test Model in its preceding MBSCT Step T2.

As shown in Figure 4.2, the entire integrated process has two *entry points*: the left-side entry point represents the start of the MBSCD process, and the right-side entry point represents the start of the MBSCT process. However, the entire SCT process has just one *exit point*: the single exit point means that any iterative/incremental development work must be tested, and the entire process is not finished until all development and testing activities have been completed.

The MBSCT methodology employs core UML diagrams in the UML subset UML–SCT (as introduced earlier in Section 3.3.2) to describe both SCD models and SCT models, enabling SCT/SCD to utilise a consistent UML-based specification approach. UML–SCT contains what we call *UML-based test diagrams*, which are mainly adapted from corresponding standard UML diagrams (e.g. UML use case diagrams, sequence diagrams, class diagrams, etc.), and are ap-

plied particularly for the purpose of UML-based SCT. In principle, a UML-based test diagram (e.g. test sequence diagram) is a diagrammatical variant of its corresponding UML diagram (e.g. sequence diagram). While a UML diagram is development-focused for describing development models in UML-based SCD, its corresponding UML-based test diagram is *testing-focused* for describing test models in UML-based SCT (e.g. illustrating relevant test operations and associated test contracts). Accordingly, the UML-based test diagrams in the UML subset UML–SCT include *test use case diagrams*, *test sequence diagrams*, *test class diagrams*, etc.

This SCT process covers a series of important UML-based SCT activities and tasks, which are further supported by other relevant MBSCT techniques. From Chapter 5 onwards, we further discuss how to put this SCT process into practice to undertake UML-based test model construction, model-based component test design and generation, component fault detection, diagnosis and localisation.

4.3.2 Scenario-Based Component Integration Testing Technique

In principle, the MBSCT methodology is developed to support SCT at various test levels /phases, including (in a bottom-up order) component operation testing, component class unit testing, component class integration testing, component integration testing (CIT) and component system testing (as shown earlier in Table 2.7 in Section 2.4). As described earlier in Section 2.4, the component user is more concerned about CIT, which is an indispensable testing phase in the SCT domain, and influences whether component reuse and integration are used correctly in producing the specified CBS. Accordingly, a principal goal of the MBSCT methodology is to support CIT, which is based on the completion of each of its underlying test levels and associated testing activities, and aims to bridge component unit testing and component system testing.

The model-based integrated SCT process (as described in Section 4.3.1) has a key focus on use case driven testing and scenario-based testing (as described earlier in Sections 3.3.2 to 3.3.3 and Sections 3.4.2 to 3.4.5), which aids in exploring the particular relationships between testing and use cases with their scenarios. A use case scenario illustrates a specific functional behaviour and forms a typical integration context covering the required interaction dynamics. Accordingly, a central CIT focus is on examining functional scenarios that specify and realise software component integration (SCI) with object interactions among integrated components and their composite objects in the specific SCI context. Using UML modeling, we can model object interactions with use cases, interaction diagrams (e.g. sequence diagrams) and class diagrams to capture scenarios, sequences, messages/operations, classes, elements (states/events), etc, which all are important testing-related component/model artefacts for CIT (which are further discussed in Section 5.3).

Consistent with use case driven principles, we incorporate scenario-based testing with CIT and conduct what we call *Scenario-Based Component Integration Testing*. One key feature of this technique is that it clearly focuses the key *CIT priority* on *test scenarios* to exercise and examine critical deliverable component functions with the associated operational use case scenarios (e.g. behavioural instances) and object interactions (e.g. integration scenarios), and to test multiple components and composite objects along with the scenario execution paths in the related SCI context. A major CIT task is to identify and construct test scenarios for the related functional scenarios that fulfil the requirements of component functions under test. Test scenarios can be represented in terms of related *test sequences* with logically-ordered *test operations* and associated *test contracts* (see Section 4.3.3 for the TbC technique). A test scenario naturally forms a useful integration testing context to examine component functions for the purpose of CIT. When applying this scenario-based CIT technique, the tester can employ a single test scenario to exercise and verify the CUT's multiple objects and operations participating in the associated scenario under test. In addition, the tester can also test a single CUT with multiple test scenarios for diverse test objectives and requirements, typically when the CUT is involved in multiple SCI contexts. Such multiple test scenarios used for CIT are especially necessary when software components are integrated into different component-based applications under development. The scenario-based CIT technique not only is consistent with the use case driven principles, but also provides a practical way to carry out use case driven testing and scenario-based testing for UML-based SCT.

One objective of the scenario-based CIT technique aims to gain a rational trade-off between test coverage and testing costs. In testing practice, full coverage testing is well known to be impractical, and high-level coverage testing also is too expensive (as described earlier in Section 2.3.5). Compared to other testing techniques, our scenario-based CIT technique prioritises test coverage by focusing on the key test scenarios that cover and verify the crucial component functions that must be delivered based on the component requirements and specifications. Such testing prioritisation is derived from the idea of identifying and using the scenario priority for core deliverable component functions, which is the primary testing concern of the component user.

The scenario-based CIT technique directly supports the MBSCT methodology's goal (4) and also partially supports the MBSCT methodology's goals (5) and (6) (as described in Section 4.2). The significance of this technique is that identifying and developing appropriate test scenarios can help establish the foundation for structuring and constructing scenario-based test models, and undertaking scenario-based test design for the CIT purpose, which will be further discussed from Chapter 5 onwards.

4.3.3 Test by Contract Technique

Based on our literature review of MBT/UBT described earlier in Chapter 3 (especially about test models in Sections 3.2.5, 3.2.6, Definition 3–2, Sections 3.3.4 and 3.5), we have introduced the new notion of "*test gaps*" and stated that developing target test models based on related ordinary software models requires bridging the identified "test gaps" between them in MBT/UBT practice. Such "test gaps" can lead to certain component artefacts being non self-testable, because the related ordinary SCD model used for describing the component artefacts under test may not comprise sufficient testing-support information and data as required for component test derivation and evaluation. For those non self-testable component/model artefacts that are required to be tested according to certain testing objectives and requirements, testing could not be properly carried out or could become very difficult to undertake effectively. In particular, when such non self-testable component/model artefacts are part of the crucial functional component scenarios under test, the associated test scenarios can become non-testable. Accordingly, using those test scenarios with poor testability for test model construction can result in ineffective SCT models being produced with poor testability. Therefore, it is necessary to develop a useful testing technique that can bridge the identified "test gaps" and improve model-based testability in MBT/UBT.

To address this second challenging research problem (as described earlier in Section 1.2 and Section 3.5), we apply the Design by Contract (DbC) [91] [92] principle to both SCD and SCT activities, and develop a novel contract-based SCT technique, *Test by Contract* (TbC). The TbC technique extends Design by Contract to the new domain of UML-based SCT, which takes it beyond the original DbC scope of code-based unit testing of traditional software classes. The TbC technique introduces a new useful testing-support mechanism of *test contracts*, which is based on the component's contract relationship between both component partners (i.e. service supplier/contractor and client) [139]. Our strategy is to leverage UML-based SCT with the test contract mechanism at different modeling levels, and design and construct adequate model-level test contracts to improve model-based component testability (see Section 2.6 for the component testability concept). The TbC technique applies well-designed test contracts to enhance and consolidate test model construction, undertake contract-based test design and generation based on test models, and conduct component fault detection and diagnosis based on contract-based tests, which are the main contract-based testing activities in the TbC technique.

Test contracts are realised and represented with preconditions, postconditions and invariants in the form of commonly-used assertions and associated concepts [152] [24], which are applied as special test conditions/constraints on particular UML model elements or component artefacts of testing interest. Based on the relationship between the *effectual contract scope* and

the software context (e.g. component context or modeling context) of a test contract, we can classify test contracts into two main categories: (a) an *internal test contract* (ITC), which has the same effectual contract scope as its software context; (b) an *external test contract* (ETC), which has the effectual contract scope different from its software context. With regard to the common usage of the two types of test contracts for SCT, an ITC is often used in component unit testing, but the ITC is required to be re-examined in the SCI context where it is involved. By contrast, an ETC is often used in CIT, where the ETC is verified in one integration module whereas it is defined and applied to another integration module. Appropriate types of test contracts are designed and constructed for improving component testability and facilitating component test design, which should be side-effect free, and should not affect or change the important sequencing attribute of relevant test sequences for model-based component test development.

The TbC technique is an integral part of the MBSCT methodology, and supports the MBSCT methodology's goals (2), (3), (5) and (6) (as described in Section 4.2). With regard to its methodological importance, the TbC technique is a primary base MBSCT methodological component, and plays the key role of a "methodology glue" that connects and incorporates the relevant MBSCT methodological components together as an integrated testing methodology to support UML-based SCT activities.

Because of the nature of the TbC technique and its special role in the MBSCT methodology, we need to create a specific layered structure and arrangement for the TbC technique contents and related technical aspects, so that we can systematically describe them in the several relevant chapters of this thesis as follows:

(1) Chapter 4 presented a basic introduction to the TbC technique (in this section).

(2) Chapter 5 applies the TbC technique to test model construction (especially in Sections 5.2.3, 5.2.4.2, 5.3, 5.4.3 and 5.5.3). This supports Phase #1 (test model construction) of the MBSCT framework (see Section 4.4 for the MBSCT Framework).

(3) Chapter 6 formally describes the TbC technique in more detail. We describe the TbC foundation principles with a set of important contract-oriented test concepts, stepwise process, test contract criteria and methods we develop for contract-based SCT (in Sections 6.2 and 6.3). On this basis, we discuss test contract design for test model construction (in Section 6.4), and contract-based component test design as well as associated technical aspects (in Section 6.5). This supports Phase #2 (component test design and generation) of the MBSCT framework.

(4) Chapter 7 particularly focuses on contract-based fault detection and diagnosis (FDD) with

the TbC technique. This jointly supports Phase #2 (component test design and generation) and Phase #3 (component test evaluation) of the MBSCT framework, enabling the MBSCT methodology to have unique contract-based and FDD-based features.

(5) Chapter 8 discusses contract-based test generation as part of component test derivation (especially in Section 8.3.2.6). This supports Phase #2 (component test design and generation) of the MBSCT framework.

4.3.4 Test-Centric Remodeling Strategy

Based on our literature review of MBT/UBT described earlier in Chapter 3 (especially in Sections 3.2.5, 3.2.6 and 3.3, and Definitions 3–1, 3–2 and 3–3), we stated that a UML-based test model should be developed based on a relevant UML-based software model. As guided by the integrated SCT process (as described in Section 4.3.1), for example, the Design Object Test Model is developed mainly based on the Object Design Model.

In practice, software models selected for test model development need to go through a form of "*remodeling*" process that is principally test-centric for transforming and improving non-testable models into testable models (as investigated earlier in Section 3.2.6.3). To support test model development effectively based on relevant ordinary software models, we introduce a new testing strategy, called *Test-Centric Remodeling* (TCR) strategy, which has three main technical aspects as follows:

(1) *Test-Centric Model Refinement*

According to the MBT/UBT principles described earlier in Chapter 3 (especially in Section 3.2.6.3), the full adoption of ordinary software models is usually not applicable in test model development, which indicates that only some model artefacts selected from ordinary software models are actually used for test model construction. On the other hand, not all artefacts from ordinary software models are useful for testing, which indicates that some model artefacts may be not required to be tested, or may be testing-irrelevant, or may not be transformable into testable artefacts to be useful for test model construction.

The TCR strategy for test-centric model refinement aims to select and retain only testing-related component/model artefacts that are useful for test model development. This means that a test model should not contain testing-irrelevant artefacts. For this purpose, we need to refine the existing software models used for test model construction by omitting redundant information from them (e.g. omitting testing-irrelevant model details, which, for example, may be concerned simply with low-level implementation aspects that may not be needed for MBT). A major goal is to require that target test models not only contain necessary testing-related component/model artefacts for test model construction, but also are rationally simpler and/or abstract than the con-

crete implementation of the component under test. This TCR strategy aspect helps test understanding and management, simplifies the test process, and eases the complexity of MBT/UBT in practice. Test models contain testing-related component/model artefacts, which form *basic test artefacts* for UML-based SCT (which will be further described in Section 5.3).

(2) *Model-Based Testability Improvement*

The TCR strategy for model-based testability improvement focuses on enhancing the selected development models (e.g. a design model) with appropriate supplementary testing-related and/or testing-support information, and transforming and improving non-testable component/model artefacts to become testable as required for constructing the target test models (e.g. a design test model). A major purpose is to improve model-based component testability for bridging the "*test gaps*" (as identified earlier in Section 3.2.6.3), and to enhance test model development for effective model-based test design, generation and evaluation.

As described in Section 4.3.3, the TbC technique is developed to support the goal of bridging the identified "test gaps" and improving model-based component testability for effective test model construction. Therefore, the MBSCT methodology supports this TCR strategy aspect by using the TbC technique to realise model-based testability improvement for UML-based SCT. Test models contain well-designed test contracts as necessary testing-support artefacts, which form *special test artefacts* for UML-based SCT (which will be further described in Section 5.3).

(3) *Test-Centric Model Optimisation*

Test-centric model optimisation builds on test-centric model refinement and model-based testability improvement. An effective approach for test-centric model optimisation is that we can construct related test models by selecting and using crucial model artefacts that have high testing priority for the principal testing objectives and requirements. A major purpose is to require target test models to contain key test artefacts with high testing priority, which accordingly supports what is a very important testing focus. This aspect of the TCR strategy helps improve and optimise test model construction to become more concise and precise for the goal of enhancing testing effectiveness.

As described in Section 4.3.2, the MBSCT methodology employs the scenario-based CIT technique that focuses the key CIT priority on identifying and developing suitable test scenarios to exercise and examine critical deliverable component functions in the related SCI context. Test scenarios are used as the primary basis to construct test models and associated model-based tests for the purpose of CIT. Accordingly, the MBSCT methodology mainly uses the scenario-based CIT technique to fulfil test-centric model optimisation.

The TCR strategy aims to provide a practical guide to carrying out actual test model development, and supports the MBSCT methodology's goals (2), (3), (4), (5) and (6) (as described in Section 4.2), in conjunction with the TbC technique and the scenario-based CIT technique. From Chapter 5 onwards, we apply the TCR strategy to test model construction for model-based test derivation.

4.3.5 Component Test Mapping Technique

Based on our literature review of MBT/UBT described earlier in Chapter 3 (especially concerning model-based tests in Sections 3.2.4 and 3.3.2.1), it is necessary to develop a testing technique for UML-based SCT that can support transforming model-based abstract test cases into concrete target test cases suitable for test execution for dynamic testing of component implementation. To further this issue, it is also required to provide practical test transformation strategies that can support the construction of model-based test artefacts with test models towards the generation of target component test cases with model-based tests to be used in the process of UML-based SCT. To address this important issue, we introduce a new mapping-based test derivation technique, called the *Component Test Mapping* (CTM) technique.

By exploring the fundamental relationships between SCD artefacts and SCT artefacts with UML models, the CTM technique creates a $(1 - n)$ test mapping relationship between the set of UML-based SCD model artefacts and the set of UML-based SCT model artefacts. That is, an element in the former set may be mapped, and thus correspond to, one or more elements in the latter set for constructing and specifying a test for a specific testing objective. The CTM technique refines and details the method and process of model-based test design and generation, and focuses on how to map and transform testable model artefacts and associated test contracts into useful test case data, so that they can be used to construct test scenarios, test sequences, test operations and test elements for generating target component test cases. The CTM process takes a series of test mapping steps for test transformations and constructions with respect to the relevant UML models and model elements at different modeling levels towards the derivation of intended component test cases. The CTM technique aids in realising test case derivation from abstract test cases to concrete test cases that are more suitable for test execution to support dynamic testing.

As an integral part of the MBSCT methodology, the CTM technique aids in carrying out the actual derivation of UML-based component test cases in UML-based SCT practice, thus the CTM technique supports the MBSCT methodology's goals (5) and (6) (as described in Section 4.2). From Chapter 5 onwards, we apply the CTM technique in conjunction with actual test model construction and component test derivation. Based on the basic introduction to the CTM

technique presented in this section, Chapter 8 further describes the CTM technique in more detail particularly concerning the test mapping definition, the CTM process and associated CTM steps towards the generation of target Component Test Specifications (see Appendix A for the definition).

4.4 MBSCT Framework

As described in Section 4.3.1, we can observe that the entire integrated process utilises the two parallel workflow streams to jointly establish an incremental and systematic framework with a series of SCD/SCT steps that covers almost all the main UML-based SCT tasks. Technically, for this MBSCT framework, we can group the related steps into four phases, as shown in Figure 4.2.

Among the four main phases, Phase #0 (including Step D0/T0) is about component selection, and not further discussed here as it is outside the scope of this research, but it is referred to elsewhere [74] [139] [66]. In addition, Step T6 in Phase #3 is about test case execution and verification with dynamic testing, which is referred to in the previous SCL work (as described in Appendix A).

Accordingly, Phase #1 to Phase #3 together form the core of the MBSCT framework (called the *three-phase testing framework*), which is described as follows:

(1) Phase #1 (including Steps D1/T1 to D4/T4): Test Model Construction – Building UML-based SCT models based on relevant UML-based SCD models (in Chapter 5)

The MBSCT methodology employs the integrated SCT process, scenario-based CIT and TbC techniques as well as the TCR strategy to construct relevant SCT models as the key foundation for UML-based SCT, which will be discussed in Chapter 5. The framework Phase #1 is model-based, process-based, scenario-based and contract-based, which is supported jointly by the first four MBSCT methodological components.

(2) Phase #2 (including Step T5): Component Test Design and Generation – Deriving component test cases from the relevant test models and other test information (in Chapter 5 to Chapter 8)

Based on related UML-based SCT models and test artefacts, the MBSCT methodology mainly employs the scenario-based CIT and TbC techniques to undertake component test design, which will be discussed in Chapter 6. Furthermore, we design component tests to detect, diagnose and locate component faults for the goal of achieving effective test design; in other

words, component fault detection and diagnosis are considered and undertaken as a crucial integral part of component test design, which will be discussed in Chapter 7.

In addition, the MBSCT methodology employs the CTM technique to refine and detail the method and process of test design and generation, especially mapping and transforming model-based abstract test cases into concrete target component test cases, which will be discussed in Chapter 8. Thus, the framework Phase #2 is model-based, process-based, scenario-based, contract-based, FDD-based and mapping-based, which is supported jointly by all of the five MBSCT methodological components.

(3) Phase #3 (including Step T7): Component test evaluation (in Chapter 7 and Chapter 9)

With model-based component tests being designed and derived, we undertake component test evaluation mainly in conjunction with the assessment of the core MBSCT testing capabilities (which are to be described in Section 4.6), which specifically focuses on validating and evaluating adequate test artefact coverage, component testability improvement, adequate component fault coverage and diagnostic solutions and results. This will be discussed in Chapter 7 and Chapter 9.

Under the MBSCT framework, Phase #1 and Phase #2 cover the important methodological aspects for developing model-based component test cases for UML-based SCT. This is an important focus of the MBSCT methodology, which provides the primary framework for Phase #3. The framework Phase #3 particularly supports the MBSCT methodology's goal (7) (as described in Section 4.2). Therefore, the five major MBSCT methodological components and the MBSCT framework jointly support all the MBSCT methodology's goals (1) to (8) (as described in Section 4.2).

4.5 Main Methodological Features

The MBSCT methodology is a comprehensive SCT approach that is jointly supported by the five major MBSCT methodological components, with the integration of new SCT/MBT concepts and definitions (as developed in Chapter 2 and Chapter 3) and commonly-used proven software concepts and practices (e.g. object-oriented methods, UML modeling, Unified Process, use case driven principles). The MBSCT methodology and its framework have their own unique methodological features different from other related work, which are:

(1) Model-based feature

The model-based feature is supported jointly by the model-based integrated SCT process, the scenario-based CIT technique and the TCR strategy. The MBSCT methodology undertakes

UML-based testing of object-oriented software components and systems, and SCT models are UML-based test models that are constructed based on relevant UML-based development models (as described in Section 4.3.1). The development of scenario-based test models emphasises the key CIT priority (as described in Section 4.3.2). The TCR strategy plays the major technical role in test model development by means of test-centric model refinement, model-based testability improvement, and test-centric model optimisation (as described in Section 4.3.4).

(2) Process-based feature

The process-based feature is characterised by the model-based integrated SCT process, which is the overall process of the MBSCT methodology (as described in Section 4.3.1). In addition, the other two major MBSCT methodological components contain their own technical processes, including the stepwise TbC working process and the CBFDD process in the TbC technique (which will be described in Chapter 6 and Chapter 7 respectively), and the stepwise CTM process in the CTM technique (which will be described in Chapter 8).

(3) Scenario-based feature

The scenario-based feature is supported by the scenario-based CIT technique that derives test scenarios and associated test sequences for UML-based CIT. This technique is consistent with the use-case driven principles, and provides a practical way to carry out use case driven testing and scenario-based testing for scenario-based test model construction and scenario-based test design (as described in Section 4.3.2).

(4) Contract-based feature

The process-based feature is characterised by the TbC technique that employs well-designed test contracts to bridge the identified “test gaps” and improve component testability for effective UML-based SCT. The TbC technique is applied to contract-based test design and evaluation, contract-based fault detection and diagnosis (as described in Section 4.3.3).

(5) FDD-based feature

The FDD-based feature is supported by the TbC technique and its CBFDD method to undertake contract-based fault detection and diagnosis, which is a primary part of component test evaluation.

(6) Mapping-based feature

The mapping-based feature is characterised by the CTM technique that maps and transforms testable UML model artefacts and special test contracts into useful test case data for generating the intended component test cases.

4.6 Core Testing Capabilities

The MBSCT methodology and its framework have six core testing capabilities that are built on the five major methodological components and the six main methodological features. The six core MBSCT testing capabilities can be classified into two main categories: the MBSCT testing applicability (including MBSCT Capabilities #1 to #3) and the MBSCT testing effectiveness (including MBSCT Capabilities #4 to #6), which are described as follows:

1. MBSCT testing applicability

(1) MBSCT Capability #1: test model construction

This testing capability is supported jointly by the first four MBSCT methodological components and the first four MBSCT methodological features.

(2) MBSCT Capability #2: component test design and generation

This testing capability is supported jointly by all five MBSCT methodological components and the six MBSCT methodological features.

(3) MBSCT Capability #3: component fault detection, diagnosis and localisation

This testing capability is supported particularly by the TbC technique and its CBFDD method, and the contract-based and FDD-based features.

2. MBSCT testing effectiveness

MBSCT testing effectiveness is based on the MBSCT testing applicability and further shows the methodological effectiveness in terms of MBSCT Capabilities #4 to #6 as follows:

(1) MBSCT Capability #4: adequate test artefact coverage

Software test artefacts designed and derived with the MBSCT methodology are capable of achieving adequate test artefact coverage of testing-related component/model artefacts and associated test contracts for the purpose of effective model-based component testing.

(2) MBSCT Capability #5: component testability improvement

Based on adequate test artefact coverage, the MBSCT methodology is capable of bridging the identified “test gaps” and improving component testability effectively for fulfilling testing requirements.

(3) MBSCT Capability #6: adequate component fault coverage and diagnostic solutions

The FDD capability is regarded as a major measure of the effectiveness of software testing approaches [33] [37]. Based on the above MBSCT Capabilities #1 to #5, the MBSCT methodology is capable of achieving adequate component fault coverage and diagnostic solutions for

the purpose of effective FDD and fulfilling testing requirements.

The first three MBSCT Capabilities #1 to #3 show the primary testing applicability of the MBSCT methodology and its framework, which will be demonstrated and validated with many selected case study examples in Chapter 5 to Chapter 8. This creates the basis for examining MBSCT testing effectiveness with the remaining three MBSCT Capabilities #4 to #6. Furthermore, this thesis employs two full case studies to undertake comprehensive validation and evaluation of the six core MBSCT testing capabilities, which will be presented in Chapter 9.

4.7 Summary

This chapter has provided an overview of the MBSCT methodology by introducing the five major MBSCT methodological components, the three-phase MBSCT framework, the six main MBSCT methodological features and the six core MBSCT testing capabilities, which form the principal original contributions of this research. Based on this overall introduction to the MBSCT methodology and its framework, we can illustrate and demonstrate how to put the MBSCT framework into practice to undertake UML-based SCT from Chapter 5 onwards. Many important technical aspects of the MBSCT techniques (especially the TbC technique, the TCR strategy and the CTM technique) will be further discussed in more detail in the subsequent chapters of this thesis in conjunction with the case studies, which validates and evaluates the applicability and effectiveness of the MBSCT methodology.

Chapter 5
Building UML-Based Test Models

5.1 Introduction

In common with MBT in general (as described earlier in Section 3.2), the MBSCT framework starts with UML-based test model development to provide the crucial foundation for UML-based SCT (including model-based component test design and evaluation). This chapter presents how to put the MBSCT methodology into practice to develop UML-based test models in the first phase of the MBSCT framework [167] [169] [171] [172] [173] [174] [175] [176] [179]. First, Section 5.2 describes the main tasks and techniques for building test models with the MBSCT methodology. Section 5.3 discusses the main test artefacts for UML-based SCT. Then, we describe the construction of the Use Case Test Model (in Section 5.4) and the Object Test Model (in Section 5.5). Section 5.6 summarises this chapter.

The testing of the Car Parking System (CPS) is the first case study that is used throughout this thesis (the CPS case study is introduced in Appendix B and is further described in Section 9.3). This chapter employs the CPS case study to illustrate (through a number of testing examples) how to apply the MBSCT methodology to the iterative and incremental development of a series of UML-based test models, with the aim to demonstrate and validate the important methodological features, applicability and effectiveness of the MBSCT methodology particularly for test model development.

5.2 Main Tasks and Techniques for Building Test Models

Following the MBSCT framework for developing model-based component test cases (as described earlier in Section 4.4), the main tasks in the first phase are to build a set of UML-based test models for SCT (i.e. SCT models) based on relevant UML-based software models for SCD (i.e. SCD models). In particular, we apply the first four MBSCT methodological components (as introduced earlier in Section 4.3) to develop UML-based test models, including the model-based integrated SCT process (in Section 5.2.1), the scenario-based CIT technique (in Section 5.2.1), the TbC technique (in Section 5.2.3), and the TCR strategy (in Section 5.2.4).

5.2.1 Applying the Model-Based Integrated SCT Process

With regard to test model development with the MBSCT methodology, the model-based inte-

grated SCT process (as described earlier in Section 4.3.1) guides what types of test models need to be constructed in terms of the different levels of use case and object test models. This entails the iterative and incremental development (which is undertaken typically with the MBSCD/MBSCT Steps D1/T1 to D4/T4) of a series of UML-based test models, including the Use Case Test Model, Analysis Object Test Model, Design Object Test Model, and Implementation Object Test Model. The integrated SCT process clearly shows what relevant SCD/SCT models are needed as the basis for constructing a specific SCT model. For example, the Object Design Model is needed as the basis for constructing the Design Object Test Model in conjunction with the Analysis Object Test Model.

A particular test model is built for a specific testing objective, for example, the Design Object Test Model is constructed mainly for the purpose of testing component objects at the design model level. This is also part of the iterative and incremental process of developing a series of test models for the purpose of deriving model-based tests from the initial form of abstract test cases towards concrete test cases (as described earlier in Section 3.2.4). Sections 5.4 and 5.5 discuss the construction of the use case test model and object test model respectively with the CPS case study.

5.2.2 Applying the Scenario-Based CIT Technique

With the MBSCT methodology, we apply the use case driven testing principle (as described earlier in Section 3.3.3) to undertake test model construction, and start with constructing the relevant use case test model, which is used to drive the iterative and incremental development of all subsequent test models. In particular, we apply the scenario-based CIT technique (as described earlier in Section 4.3.2), and undertake the construction of a target test model for the purpose of UML-based CIT (which is a principal goal of the MBSCT methodology). Our testing priority focuses on identifying and constructing appropriate *test scenarios* with relevant operational use case scenarios (e.g. behavioural instances and integration scenarios) to exercise and examine crucial deliverable component functions, and to test multiple components and composite objects along with the scenario execution paths in the associated SCI context. We model test scenarios mainly with scenario mapping and transformations from the corresponding functional scenarios under test, which are usually described with UML use case diagrams, sequence diagrams and class diagrams in the relevant UML-based SCD models. Accordingly, test scenarios are typically captured with *test use case diagrams*, *test sequence diagrams* and *test class diagrams*, which are the main UML-based test diagrams (in the core UML subset UML–SCT) that are used in the MBSCT methodology to describe UML-based test models (as described earlier in Section 4.3.1 and Section 3.3.2). Test scenarios are specified in terms of relevant *test sequences* consisting of logically-ordered *test operations* and associated *test contracts*, which aid

in structuring and constructing relevant scenario-based test models and scenario-based test design for UML-based SCT. Relevant illustrative examples are described with the CPS case study in Sections 5.4.2 and 5.5.2.

5.2.3 Applying the TbC Technique

As investigated earlier in Section 3.2.6.3 and in Section 4.3.3, the presence of "*test gaps*" is a major cause of the production of ineffective test models with inadequate testability. Good test model development is required to improve model-based component testability by means of transforming and enhancing non-testable component/model artefacts under test to become testable as required (e.g. for the target testing objectives and requirements). To bridge the identified "test gaps" for effective test model construction, we apply the TbC technique (as described earlier in Section 4.3.3) to design and construct useful test contracts as a key testing-support mechanism for improving model-based component testability. We augment the relevant test models under development with well-designed test contracts for model-based test enhancements. When constructing a particular SCT model, we can incorporate appropriate test contracts with the relevant artefacts of components and/or their composite classes under test. For example, test contracts can be used as preconditions/postconditions to complement the component/class operation under test; similarly, test contracts can be also used as invariants to complement the component/class under test. By means of such model-based test improvement, we can effectively transform non self-testable operations or similar component/model artefacts under test to become testable, and accordingly, we can enhance and consolidate test model construction to achieve good component testability. Relevant illustrative examples are provided with the CPS case study in Sections 5.4.3 and 5.5.3.

Based on contract-based testability enhancement for effective test model development, the TbC technique can further support UML-based SCT for model-based component test design and generation, and component fault detection, diagnosis and localisation. This will be further discussed in Chapter 6 and Chapter 7, which formally describe the TbC technique and associated technical aspects in more detail.

5.2.4 Applying the TCR Strategy

As described earlier in Section 3.2.6.3 and Section 4.3.4, test model development requires a "remodeling" process that transforms and improves relevant ordinary SCD models (which are not test-ready or are non-testable) into target SCT models (which are test-ready or testable). With the MBSCT methodology, we apply the TCR strategy to create such a test-centric remodeling process for test model construction by means of test-centric model refinement, model-

based testability improvement, and test-centric model optimisation (as earlier described in Section 4.3.4). The TCR strategy plays a major technical role in test model construction, which is carried out in cooperation with the application of the relevant MBSCT techniques.

5.2.4.1 Test-Centric Model Refinement

Model-based component testing deals with both *component artefacts* in component software being implemented and *model artefacts* in models describing relevant component artefacts under test. It is important to recognise not only what component/model artefacts are *testing-required* (i.e. they are required to be tested, are needed for testing, and/or can be used for testing purposes), but also what component/model artefacts are capable of being self-tested or are *self-testable*. The *testing-related* component/model artefacts comprise these testing-required and testable artefacts. A major purpose is to utilise such testing-related component/model artefacts to effectively support test model development and model-based test derivation. To carry out test-centric model refinement with the TCR strategy (as earlier described in Section 4.3.4), a key challenge is how to identify and extract (and/or to design and construct, if needed) testing-related model artefacts in relevant SCD models that are useful to construct corresponding SCT models.

For the purpose of UML-based CIT (which is a principal goal of the MBSCT methodology), our approach focuses on a range of core testing-related component/model artefacts and elements, which are described as follows:

(1) *Use Case Model: use case diagrams and use cases* (see Sections 3.3.2 and 3.3.3)

These model artefacts describe component system/integration requirements and behavioural functions in terms of use case specifications. We focus on identifying and extracting the main testing-related model artefacts, including system/integration-level use cases with their scenarios, system sequences, system events/operations, system states, etc. They are most important to derive system/integration test requirements and objectives, and use-case based test scenarios and associated test artefacts, which forms the basis for use case driven testing. Section 5.4 discusses in detail how these use case model artefacts are used to construct the target use case test model.

(2) *Behavioural Model: sequence diagrams and interacting messages* (see Section 3.3.2)

These model artefacts comprise the dynamic models to capture integration dynamics, and describe how a use case scenario (e.g. for specifying a behavioural instance or an integration function) is realised and how interactions are conducted with a sequence of interacting messages over time between collaborating components/objects. The main testing-related model artefacts

we can identify and extract include concrete scenarios refining and realising use cases, message sequences describing integration interactions, interacting messages describing collaborations, software classes/objects participating integration/interactions, class operations/states realising messages, etc. Section 5.5 discusses in detail how these model artefacts are used to construct the related object test model.

(3) *Structural Model: class diagrams, operations and elements* (see Section 3.3.2)

These model artefacts comprise the static models to provide the structure of software components and systems under test. They define software classes (e.g. operations, states and attributes), and describe class interfaces and their relationships, which are testing-related and provide the essential test information and data for test model construction (to be discussed in detail in Section 5.5).

By applying the TCR strategy for test-centric model refinement (as discussed above), we can develop the required core testing-related component/model artefacts that are identified and extracted from the relevant UML-based SCD models, which form the principal foundation for test model construction. A primary goal of the test-centric model refinement strategy is to ensure that test models do not include redundant testing-irrelevant information, so that the target test models are test-focused, and are simpler and more abstract than the component implementation under test.

5.2.4.2 Model-Based Testability Improvement

In addition to the required core testing-related component/model artefacts being developed, we need to apply the TCR strategy for model-based component testability improvement (as described earlier in Section 4.3.4) to design and construct supplementary test artefacts as required, in order to bridge the identified "test gaps" in UML-based SCT for achieving the desired testing effectiveness. The notion of the "*test gaps*" was initially introduced in Section 3.2.6.3 and described in Sections 4.3.3 and 5.2.3, and we have stated that the occurrence of "*test gaps*" is a major cause of inadequate model-based testability. This section further analyses and explores its underlying attributes and associated issues, and discusses how to apply the TCR strategy to deal with them for test model construction with effective testability improvement.

We focus on two main types of "test gaps" for mode-based testability improvement as follows:

(1) *Bridging Test-Gap #1 with Supplementary Testing-Related Component/Model Artefacts*

There are some situations where the existing testing-related component/model artefacts in

the associated ordinary SCD model are insufficient or incomplete for the purpose of test model construction and model-based test development. This occurs especially when the relevant SCD model leaves out some important testing-related information (e.g. relevant component/model artefacts) that is required for developing appropriate test scenarios, test sequences, test operations or other related test artefacts in the test model under development. Although the omission of such testing-related component/model artefacts may not affect component design and/or implementation, the absence of these testing-related model artefacts could lead to a *failure* to adequately describe some aspect or the whole of a particular testing-required component artefact (e.g. a component/class operation under test) for testing purposes. As a negative consequence, this could further result in the *subsequent failure* to exercise and examine the related component artefact (e.g. failing test execution of the testing-required component/class operation) for the target testing objective and requirement. Accordingly, a particular type of "*test gap*" results from the omission of such testing-related component/model artefacts if they are required to be tested.

To deal with this first type of "test gap" (we call it *Test-Gap #1*) for enhancing testing effectiveness, we need to design and construct appropriate supplementary testing-related component/model artefacts (which are testing-required or are testable), and add these relevant test artefacts to the test model under development. This is consistent with the principle of model-based testability improvement with the TCR strategy as a major purpose is to develop appropriate testing-related component/model artefacts that are sufficiently adequate for the target testing objectives and requirements. In practice, how to design and construct appropriate supplementary testing-related component/model artefacts in the form of additional test artefacts is based on several aspects, including: the component requirements and specifications, the target testing objectives and requirements to be achieved, the tester's knowledge of the associated SCD model actually used for test model development, the tester's testing skills and experience, etc. This resembles the similar situation of how to carry out improvement of effective SCD in CBSE practice. It is very difficult or even impractical to exercise and examine certain testing-required but omitted component/model artefacts for testing purposes without such supplementary testing-related artefacts. A relevant illustrative example is given with the CPS case study in Section 5.5.2.

(2) *Bridging Test-Gap #2 with Complementary Testing-Support Artefacts* (*Test Contracts*)

The combination of the existing and supplementary testing-related component/model artefacts can jointly form the prototype of the test model with adequate testing-related artefacts for the purpose of UML-based SCT. Then, according to certain testing objectives and requirements, we need to undertake special treatment for certain testing-related component/models artefacts under test, if they are required to be tested, but they are *not self-testable*, i.e. such testing-related but non-testable model artefacts could not be used as the sole basis for properly testing the asso-

ciated component artefact (e.g. a component/class operation under test) that is merely described by them. Accordingly, another type of "*test gap*" results from the inadequate testing capability (i.e. inadequate testability) of such non-testable component/model artefacts if they are required to be tested.

To cope with this second type of "test gap" (we call it *Test-Gap #2*), we need to transform and enhance those non-testable component/model artefacts to become testable by means of model-based testability improvement, which is realised by applying the TbC technique (as described in Section 4.3.3 and Section 5.2.3). Well-designed test contracts can provide additional useful testing-support information and data to complement the relevant test artefacts for the test model under development, so that we can transform and enhance non-testable component/model artefacts under test to be testable as required for UML-based SCT. For example, a test contract (e.g. in the form of a postcondition assertion) is constructed and then applied to a specific component/class operation under test to verify (e.g. by checking test results) whether this operation is performed correctly against its component functional requirement. It is extremely difficult or even impossible to examine and evaluate the actual model-based test execution of those testing-related but non-testable component/class operations without such complementary testing-support artefacts. A relevant illustrative example is described with the CPS case study in Section 5.5.3.

(3) *Bridging Both Test-Gap #1 and Test-Gap #2 to Improve Component Testability*

Note that there are some important implications concerning these two types of "*test gaps*". Test-Gap #1 is caused by the omission of certain component/model artefacts that are testing-related and required to be tested, and thus we need appropriate supplementary testing-related artefacts for testing purposes. Test-Gap #2 is caused by the inadequate testability of certain testing-related component/model artefacts that are required to be tested, but are not self-testable, and thus we need appropriate complementary testing-support artefacts for testing purposes.

From the current literature review, there is very little research work on dealing with Test-Gap #1, which may well be based on an implicit *assumption/misconception* in MBD/MBT: all necessary testing-related information (e.g. basic test artefacts) are available in software development models for all testing purposes. Likewise, the occurrence of Test-Gap #2 may well be due to another similar implicit *assumption/misconception* in MBD/MBT: all testing-related artefacts available in software development models are testable for all testing purposes. However, both assumptions are not always valid, because in practice there is no perfect software development model that can fulfil such extraordinary testing-centric requirements. We can observe that simply bridging Test-Gap #1 would not always ensure that the target testing objective is accomplished successfully, and bridging Test-Gap #2 is actually more important in test model

construction for effective model-based testing. Therefore, it is very important to bridge both Test-Gap #1 and Test-Gap #2 to improve model-based component testability, in order to achieve the target testing objectives and desired testing effectiveness.

5.2.4.3 Test-Centric Model Optimisation

By using the TCR strategy for test-centric model refinement and model-based testability improvement, we can develop useful test artefacts (including testing-related component/models artefacts and associated testing-support artefacts) to construct test models for UML-based SCT. Furthermore, we can improve and optimise test model construction to prioritise on the most important test artefacts by means of test-centric model optimisation with the TCR strategy (as described earlier in Section 4.3.4). For the purpose of UML-based CIT, our testing priority focuses on appropriate test scenarios to exercise and examine core integration scenarios with multiple integrated components and composite objects in the associated SCI contexts. Such SCI-related test scenarios can be used as the foundation for structuring and constructing relevant scenario-based test models and scenario-based test design for the CIT purpose, which is supported by applying the scenario-based CIT technique (as described in Section 4.3.2 and Section 5.2.2).

5.2.5 Summary

As discussed in the above Sections 5.2.1 to 5.2.4 (including Subsections 5.2.4.1 to 5.2.4.3), we can observe that the TCR strategy plays the major technical role, and incorporating it with the related MBSCT techniques can effectively guide test model development in UML-based SCT practice. Sections 5.4 and 5.5 will employ the CPS case study to illustrate by examples the important methodological characteristics and technical aspects of the MBSCT methodology on test model construction.

As a summary, to construct a UML-based SCT model (e.g. the design object test model) based on its related UML-based SCD model (e.g. the object design model), we need to carry out the following tasks with the MBSCT methodology (e.g. the TCR strategy and the related MBSCT techniques):

(1) Applying test-centric model refinement with the TCR strategy (as discussed in Section 5.2.4.1):

We identify and extract the core existing testing-related component/model artefacts from the related UML-based SCD model, and transform and enhance them to become appropriate *basic test artefacts* (see Section 5.3).

(2) Applying model-based testability improvement with the TCR strategy:

(a) If the testing-related artefacts (which are mainly used for basic test artefacts) in the associated ordinary SCD model are insufficient or incomplete (this is Test-Gap #1 as discussed in Section 5.2.4.2):

We need to design and construct certain supplementary testing-related component/model artefacts, and appropriately add these basic test artefacts to the test model under development.

(b) If some testing-related artefacts (as basic test artefacts) are not self-testable (this is Test-Gap #2 as discussed in Sections 5.2.4.2 and 5.2.3):

We need to design and construct certain complementary testing-support artefacts (e.g. special test contracts), transform and enhance non-testable component/model artefacts under test to be testable as required, and then appropriately add these *special test artefacts* (see Section 5.3) to the test model under development. This is carried out in conjunction with applying the TbC technique.

(3) Applying test-centric model optimisation with the TCR strategy (as discussed in Sections 5.2.4.3 and 5.2.2):

We can improve and optimise test model construction by focusing our testing priority on core SCI-related test scenarios as the primary basis to structure and construct relevant test models for the CIT purpose. This is carried out in conjunction with applying the scenario-based CIT technique.

5.3 Test Artefacts for UML-Based SCT

During the course of test model development with the MBSCT methodology, we identify, extract, design and construct a range of useful test artefacts that correspond to testing-related component/model artefacts and associated testing-support artefacts (as described in Section 5.2). Typical test artefacts used for UML-based SCT mainly include test use cases, test scenarios, test sequences, test messages, test operations, test classes/objects, test elements (e.g. test states, test events), and special test contracts, while some additional test artefacts may also be needed depending on the specific testing requirement or environment used in testing. We can classify relevant test artefacts into two main categories: basic test artefacts and special test artefacts (as shown in Table 5.1), which work together in UML-based SCT.

(1) *Basic Test Artefacts*

These test artefacts are built on the core existing *testing-related component/model artefacts* and elements that are testing-required or are testable, which is carried out mainly with the TCR strategy for test-centric model refinement (as described in Section 5.2.4.1). They are iden-

tified and extracted based on the corresponding SCD models, and are then transformed into basic test artefacts in test model construction to exercise and examine component functions with operational scenarios and/or related component/model artefacts for UML-based SCT. In addition, we need to design and construct certain supplementary testing-related component/model artefacts for enhancing testing effectiveness, and add these useful test artefacts to the test model under development (as described in Section 5.2.4.2).

Under this category, there are several types of basic test artefacts being produced in terms of the granularity of test artefacts, which are summarised in Table 5.1. These basic test artefacts principally form the prototype of the test model under development.

Table 5.1 Test Artefacts for UML-Based SCT

Test Artefact	Description	Test Level
Test Use Case	A test use case exercises and examines one or more related use cases (e.g. behavioural use cases) under test, and is usually structured into use-case related test sequences.	Integration /System Testing
Test Scenario	A test scenario exercises and examines one or more related use case instances (e.g. behavioural scenarios) under test, and is usually structured into scenario-related test sequences. A test scenario is a particular instance of its corresponding test use case.	Integration /System Testing
Test Sequence	A test sequence consists of a sequence of logically-ordered test messages, test operations and/or other related test artefacts.	Integration /System Testing
Test Message	A test message exercises and examines the corresponding message(s) under test for verifying relevant message-based interactions between collaborating components/objects.	Integration /System Testing
Test Event	A test event exercises and examines the corresponding event(s) under test for relevant event-based communications between collaborating components /objects. It represents the special test message(s) that take the form of event.	Integration /System Testing
Test Operation	A test operation is used to exercise and examine the corresponding operation(s) under test for component/class operation testing. Test operations are essentially used for unit testing. In addition, test operations also realise the related test messages/events and relate to component system/integration testing. Thus, they support all test levels.	Unit Testing, supporting all test levels
Test State	A test state is used to exercise and examine the corresponding state(s) under test that reflects the current condition/situation or change of its host class /object (e.g. values of class/object attributes). Test states provide the essential test information that relates to and supports all test levels.	Supporting all test levels
Test Class	A test class is used to exercise and examine the corresponding class(s) under test. Test classes provide the essential test information and data that relate to and support all test levels.	Supporting all test levels
Test Contract	A test contract provides additional testing-support information and data to complement the relevant test artefacts, transforming and enhancing non-testable component/model artefacts under test to become testable as required.	Supporting all test levels

(2) *Special Test Artefacts*

Special test artefacts are designed and constructed to improve model-based component

testability with the TCR strategy for effective test model development (as described in Section 5.2.4.2). These test artefacts are mainly composed of complementary testing-support artefacts (e.g. special test contracts as shown in Table 5.1), which aid testing-related component/model artefacts under test to become testable if they are not self-testable.

Note that there is a major difference here: an ordinary testing-related operation (as a basic test artefact) essentially exercises the execution of its relevant component function(s), whereas its associated test contract (as a special test artefact) employs appropriate testing-support assertions to verify whether the operation execution is correct and complies with the expected requirement. This is because test contracts with testable assertions can be used to design *test oracles* for verifying the expected test results. Moreover, if the associated test contract returns *false*, a possible component fault is then detected. We can see that such test contracts as special test artefacts can well improve component testability for effective UML-based SCT. Test contracts and contract-based fault detection and diagnosis with the TbC technique will be described in more detail respectively in Chapter 6 and Chapter 7, in conjunction with relevant illustrative examples selected from the CPS case study.

For UML-based SCT with the MBSCT methodology, test models mainly contain basic test artefacts (e.g. the core existing and supplementary testing-related component/model artefacts that are testing-required or are testable) and special test artefacts (e.g. the special test contracts as complementary testing-support artefacts that enable non-testable component/model artefacts under test to become testable). Test models do not need to, and should not, include other redundant testing-irrelevant artefacts as required by the TCR strategy for test-centric model refinement (as described in Section 5.2.4.1). Both basic and special test artefacts jointly work to undertake UML-based SCT. A complex test artefact often takes the form of a combination of both basic and special test artefacts. For example, a test scenario is a sequence of logically-ordered test messages, test operations and/or associated test contracts.

5.4 Use Case Test Model

The preceding Sections 5.2 and 5.3 have presented the important technical aspects of applying the MBSCT methodology to test model development. On this foundation for test model development, we are able to construct individual UML-based SCT models in the MBSCT Steps D1/T1 to D4/T4 (as described in the integrated SCT process in Section 4.3.1 and Section 5.2.1). The following Sections 5.4 and 5.5 focus on the particular technical aspects for constructing a specific test model in a MBSCT step. We will employ the CPS case study to illustrate by examples the relevant technical aspects for test model construction with the MBSCT methodology particularly for the CIT purpose (as indicated in Section 5.1).

The model-based integrated SCT process requires that there are two major levels of test models under development: Use Case Test Model (UCTM) and Object Test Model. This section discusses the first MBSCT Step: D1 → T1 to construct the UCTM mainly based on the related Use Case Model (UCM) at the use case level for the CIT purpose.

5.4.1 Constructing the Use Case Test Model

Using UML models, the UCM mainly describes the system/integration behaviour, functions and requirements in terms of a set of actors (e.g. component users), use cases and their relationships as well as use case specifications for the CBS (component-based system) under test (as described earlier in Section 3.3.2). Our main task is to focus on identifying and extracting, designing and constructing testable component/model artefacts with the UCM, and then transforming and enhancing them to become appropriate test artefacts for constructing the UCTM (as shown in Figure 5.1). The UCTM is mainly described with test use case diagrams and system test sequence diagrams in the core UML subset UML–SCT (as shown in Figure 5.2).

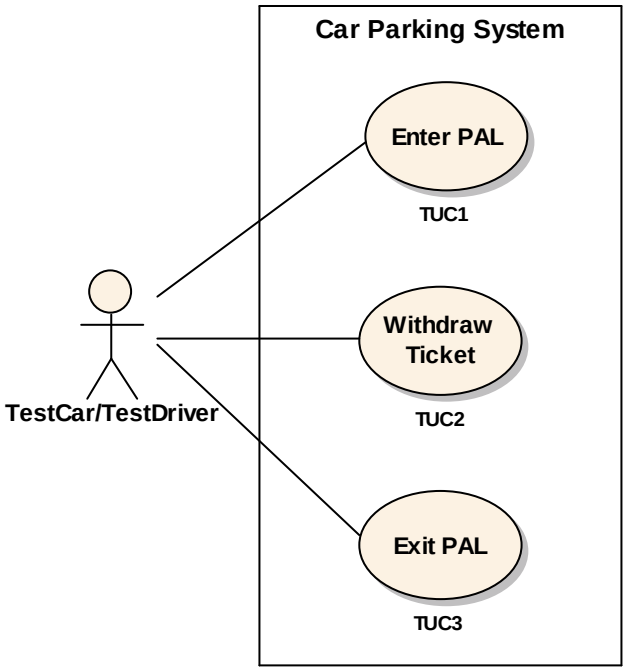
We apply the TCR strategy to develop basic test artefacts for establishing the prototype of the UCTM (as described in Section 5.2.4). We further use some selected examples of the CPS case study to illustrate how to develop SCI-related test scenarios and test contracts for the UCTM construction in the following subsections (in Sections 5.4.2 and 5.4.3).

D1: Use Case Model	T1: Use Case Test Model
1. Functions and requirements.	1. Testing objectives and requirements.
2. Use-case diagrams.	2. Test use case diagrams with test actors, test events.
3. Actors and descriptions.	3. Test actors and descriptions.
4. Use cases and scenario descriptions.	4. Test use cases and test scenario descriptions.
5. System sequence diagrams for system scenarios with system events.	5. System test sequence diagrams for system test scenarios with test actors, test events.
6. Contracts for system events and scenarios.	6. Test contracts for system test events and scenarios.

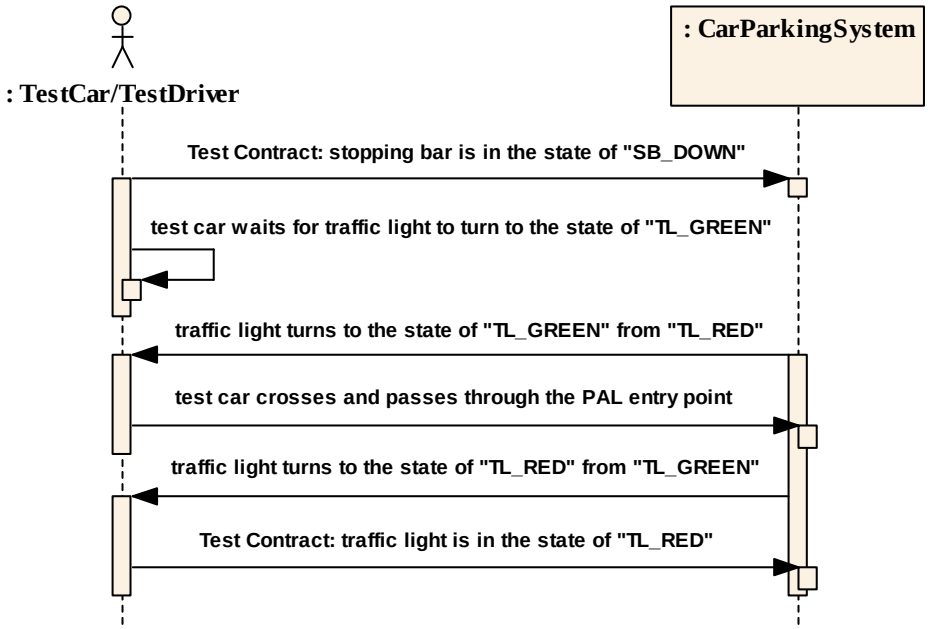
Figure 5.1 Constructing the Use Case Test Model

5.4.2 Identifying and Constructing Test Scenarios

We apply the scenario-based CIT technique to identify and construct relevant test use cases and test scenarios that have high testing priority as the primary basis for developing the UCTM (as described in Sections 4.3.2, 5.2.2 and 5.2.4.3). For the CIT purpose, test scenarios are developed based on the associated use case instances to exercise and examine the corresponding SCI scenarios that fulfil component functions in the SCI context.



(a) Test Use Case Diagram (CPS System)



(b) System Test Sequence Diagram (CPS TUC1 Test Scenario)

Figure 5.2 Use Case Test Model (CPS System)

Among other CPS use cases, we identify and construct three core *test use cases* (TUCs) to develop test scenarios for testing typical CPS operations:

(a) TUC1: exercise and examine that the test car enters the entry point of the parking access

lane (PAL) to start accessing the PAL;

(b) TUC2: exercise and examine that the test driver withdraws a parking ticket at the PAL ticket point;

(c) TUC3: exercise and examine that the test car exits the PAL exit point to finish accessing the PAL.

All TUCs for these three main parking phases constitute an overall test scenario/sequence of one full parking access process cycle for any parking in the CPS system. Because the car movement along the PAL interacts with a set of the CPS parking control devices, each of the TUC test scenarios conducts certain CIT activities to exercise and examine the relevant CPS operations. These TUCs provide the typical CIT contexts to verify the related integration test scenarios. As an example, Figure 5.2 shows a partial UCTM of the CPS system, with a test use case diagram for the three core TUCs (see Figure 5.2 (a)) and a system test sequence diagram that illustrates the system test scenario for the first CPS TUC1 test scenario (see Figure 5.2 (b)).

With the UCTM, a *test actor* plays the representative testing role of the users of use cases of the CBS under test. For the CPS system, a test actor is a *test car* (or equivalently, a *test driver* of the car) that represents the CPS user that is eligible to access the PAL for car parking. A *system test event* exercises and examines related system events (e.g. parking control operational activities) that cause an interaction between the test actor and the system. A *test scenario* is a typical test use case instance (e.g. an instance of TUC1 in Figure 5.2 (b)), which exercises and examines a sequence of system test events that occur between the test actor and the black-box system under test (e.g. our CPS system), and thus tests the associated system operational use case scenarios for the required behaviour (e.g. the test car enters the PAL correctly) in the use case under test (e.g. TUC1). A test scenario is captured with a system test sequence diagram to illustrate the corresponding scenario-based UCTM (as shown in Figure 5.2).

5.4.3 Designing and Constructing Test Contracts

In the UCTM, a test scenario also reflects the corresponding changes of relevant *system test states* (e.g. the traffic light turns the state of "TL_GREEN" from "TL_RED" or vice versa, as shown in Figure 5.2 (b)), which are usually trigged by system test events (e.g. parking control operational activities) and are very useful in scenario-based testing. A clear testing objective is that certain functional requirements (e.g. the test car should enter the PAL correctly in the TUC1 context) are correctly fulfilled as expected through an examination of the related test scenario and associated test states (e.g. "TL_GREEN", "TL_RED" in the TUC1 test scenario).

Because the UCTM treats the entire CBS as a black-box entity at the system test level, the main test contracts developed with the TbC technique for the system-level scenario under test

consist of a set of system-level preconditions, postconditions and invariants, which are special test artefacts in the related test scenario for constructing the UCTM. The system-level test contracts are used to examine and verify conformance to the testing requirements in the related system-level test scenario. Taking the CPS TUC1 test scenario as a testing example, we can design and construct the following system-level test contracts (as shown in Figure 5.2 (b)):

(1) *TUC1 preconditions*:

(a) All CPS control device modules are started and are in an operational status;

(b) The test car is started, ready and eligible to access the PAL;

(c) The stopping bar is in the state of "SB_DOWN", after the last car has finished access and exited the PAL in the last parking access process cycle, and before the new car enters the PAL. This partially abides by the special mandatory parking assess safety rule in the CPS system: "*one access at a time*" (which is one of the CPS special test requirements to be described in Section 9.3.1 for the full CPS case study);

(d) The traffic light is in the state of "TL_GREEN", before the test car starts entering the PAL.

(2) *TUC1 postconditions*:

(a) The test car has entered the PAL;

(b) The traffic light is in the state of "TL_RED", after the current car has entered the PAL. This also partially abides by the same special safety rule: "*one access at a time*".

(3) *TUC1 invariants*:

The abovementioned safety rule ("*one access at a time*") is a typical invariant, which is applied to and required for all parking control operations and car parking activities in the CPS system.

Note that, because the UCTM is built with regard to the black-box system under test in the first MBSCT Step D1 → T1, certain internal system event/state changes may be invisible to the external actor in the UCTM (e.g. the state changes of the in-PhotoCell sensor device that monitors cars entering into the PAL, which are internal to the CPS system). Such internal operation information and relevant test artefacts will be further explored and illustrated in subsequent test models (see Section 5.5.2). The UCTM is the initial step in test model construction, which leverages system level scenarios to develop a set of core test scenarios for scenario-based CIT. The UCTM describes the main test requirements with associated test scenarios and test contracts, which form the basis for use case driven testing to guide the stepwise testing activities towards the iterative and incremental development of subsequent test models with concrete and detailed test artefacts.

5.5 Object Test Model

Working with object-oriented testing techniques for test model development at the object test level, we build a series of object test models, including the Analysis Object Test Model, Design Object Test Model, and Implementation Object Test Model. Object test model development is undertaken in the MBSCT Steps D2/T2 to D4/T4, which follow different object-oriented development phases that require different levels of class/object details. This section focuses on the MBSCT Step D3 → T3 to describe the construction of the Design Object Test Model (DOTM) based on the Object Design Model (ODM), which serves as an example of test model development at the object test level. Our primary purpose here is to use the DOTM as a representative test model to undertake UML-based CIT.

5.5.1 Constructing the Object Test Model

The UML-based object model captures and specifies component-based systems in terms of objects/classes (attributes, operations) and their relationships (associations, interactions, collaborations), and its structure is represented with UML class diagrams (as described earlier in Section 3.3.2). We base CIT on the behavioural object model (e.g. the ODM) that describes the use case realisation for dynamic behaviour and functions in terms of collaborating objects and their interactions in the related SCI context, which is typically represented with UML sequence diagrams (as described earlier in Section 3.3.2).

To develop the corresponding object test model with the MBSCT methodology, we first develop the basic test artefacts to produce the prototype of the object test model with the TCR strategy (as described in Section 5.2.4 and Section 5.3). Our main tasks are to identify and extract, design and construct testable component/model artefacts with the related object model (e.g. the ODM), and then transform and enhance them to become useful test artefacts for constructing the target object test model (e.g. the DOTM). Then, we apply the TCR strategy and related MBSCT techniques, and employ some selected examples of the CPS case study to illustrate how to undertake test-centric model optimisation with crucial SCI-related test scenarios and how to undertake model-based testability improvement with well-designed test contracts for effectively constructing the DOTM (this process is to be further described in the following Sections 5.5.2 and 5.5.3). As a typical illustration of test model development at the object test level, Figure 5.3 shows constructing the DOTM mainly based on the related ODM in the MBSCT Step D3 → T3. The object test model can be represented with test class diagrams and test sequence diagrams in the core UML subset UML–SCT (as shown in Figure 5.4).

Note that there is a major difference here between the UCTM and DOTM: test artefacts in the object test model correlate now with relevant test classes and associated test elements, rather

than to the entire black-box system at the use case level. For example, test artefacts in the DOTM can be specified and represented with design test classes that are developed based on relevant design classes in the ODM, in conjunction with certain supplementary testable component/model artefacts (as shown in Figure 5.3). Furthermore, some internal operation information and associated test artefacts of the CBS under test can be explored and tested by relevant class elements with the DOTM.

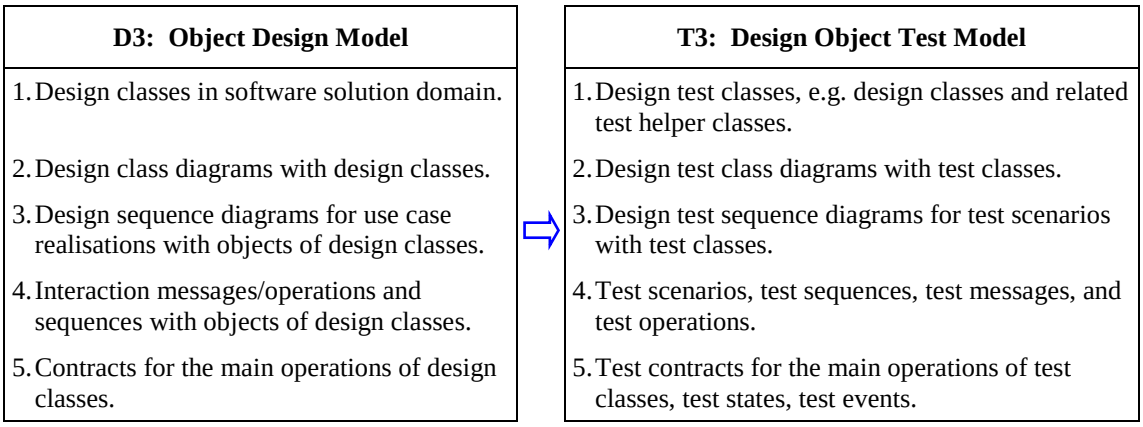


Figure 5.3 Constructing the Design Object Test Model

5.5.2 Test Scenarios for Test Model Construction

As a SCT model for testing design objects, the DOTM is constructed with test scenarios, test sequences, test messages, test operations, and test classes as well as test contracts at the object design level. Figure 5.4 shows a design test sequence diagram for the first CPS TUC1 test scenario, which is part of the DOTM for testing the CPS system. We intend to perform CIT on how the test car enters the PAL correctly in the TUC1 integration testing context, where the PAL entry point is jointly controlled by the traffic light and in-PhotoCell sensor devices. For this CIT purpose, we apply the scenario-based CIT technique to develop the corresponding test scenario: exercising and examining the crucial object interactions with the associated integration-participating operations and associated test artefacts with relevant test classes in the CIT context. As shown in Figure 5.4, we construct the DOTM based on the TUC1 test scenario to verify the related parking control operations of how the test car enters the PAL correctly in TUC1. The TUC1 test messages for verifying object interactions can be realised with the associated integration-participating operations and associated test artefacts, which are described with six relevant test objects/classes (e.g. two of these are class `TrafficLight` in the device control component and test object `testCarController` in the car control component). Test scenarios (e.g. the CPS TUC1) establish the basic structural framework for the test model under construction (e.g. the CPS DOTM) in terms of crucial test sequences that are composed of the logically-ordered test operations from the related test classes and complementary test contracts added to the test classes.

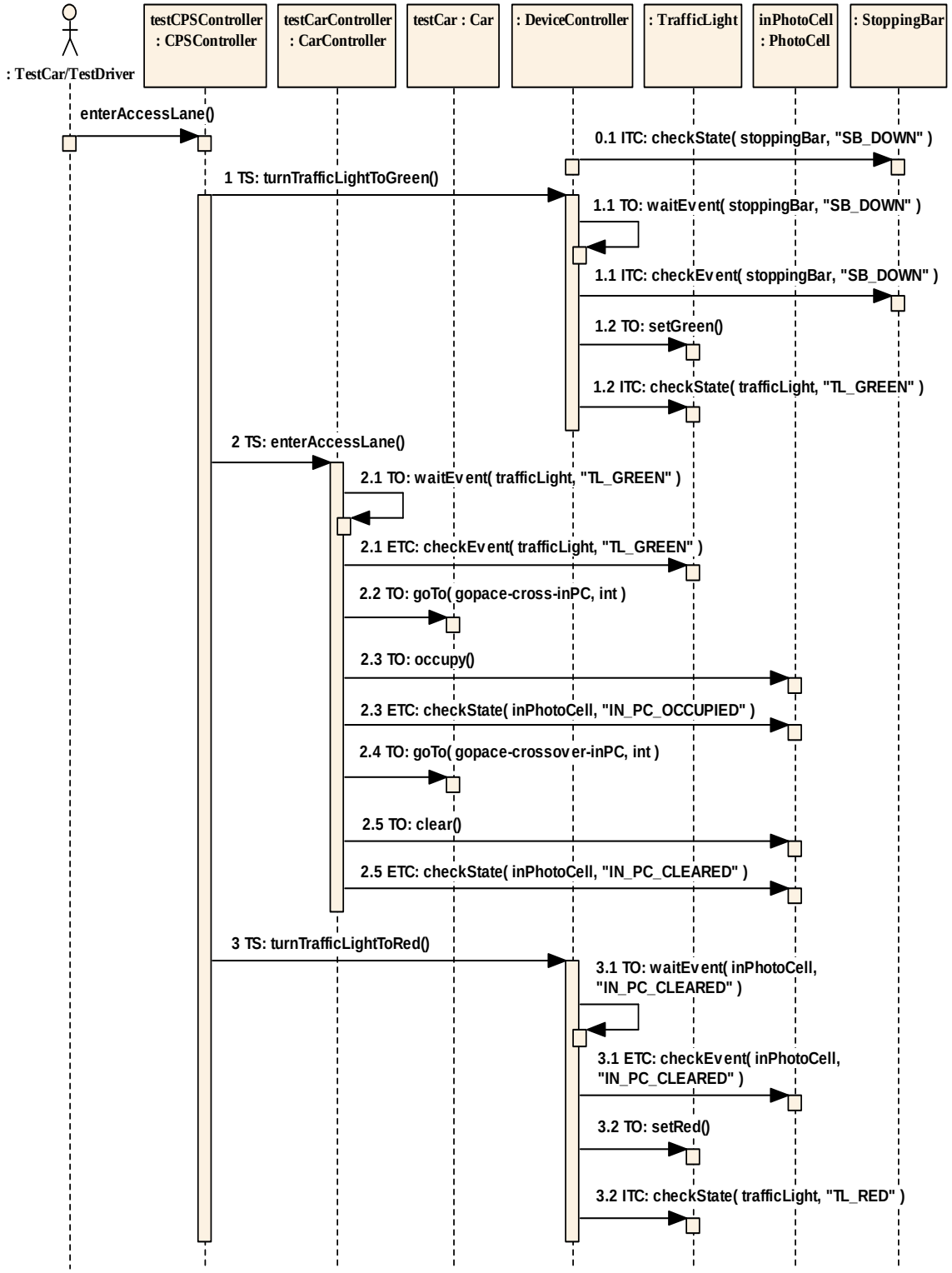


Figure 5.4 Design Object Test Model (CPS System)

Design Test Sequence Diagram (CPS TUC1 Test Scenario)

With test scenarios for constructing the DOTM, we use the car control component for the test car to interact with the CPS system under test, and for the CIT purpose, we examine:

(a) Whether the parking control operations function correctly with the parking control devices in the device control component;

(b) Whether the test car correctly performs its parking access to the PAL;

(c) Whether the CPS operations properly abide by the mandatory parking assess requirements (e.g. the special parking assess safety rule: "*one access at a time*").

For the CIT purpose with the DOTM, test messages for verifying object interactions are mainly realised and represented with related test operations and test contracts. For example, a basic test operation (e.g. `setGreen`()) from its test class (e.g. class `TrafficLight`) exercises and examines what the CPS system does with the operation under test (e.g. the traffic light is set to the state of "TL_GREEN"). Besides testing the external operations (e.g. `setGreen`()) visible to the CPS user (e.g. the car/driver), we can now describe and examine the CPS internal operations with relevant test class elements in the DOTM. For example, operation `occupy`() is performed internally inside the TUC1 scenario, where the in-PhotoCell sensor device monitors and detects whether the test car occupies and crosses the entry point in the PAL. This CPS internal control operation and its associated state information, which were invisible to the external car/driver in the UCTM at the use case level (as described in Section 5.4.3), can now be exercised and examined with test class `PhotoCell` in the DOTM. This example demonstrates how more detailed component artefacts in the CBS under test can be explored and tested with the MBSCT methodology at the MBSCT Step D3 → T3 for the DOTM construction. Similar testing tasks are also undertaken at each related MBSCT Step, as the integrated SCT process advances forward iteratively and incrementally (as shown in Figure 4.2).

In addition to the existing basic test artefacts identified and extracted from the current ODM, we also need to supplement certain testing-related artefacts to the associated test scenario for constructing the scenario-based DOTM, in order to bridge *Test-Gap #1* as described in Section 5.2.4.2 (1). For example, suppose the current ODM is not adequate and has left out operation `setRed`(). Since this operation is performed to set the traffic light to the state of "TL_RED" only after the car enters into the PAL, it is not actually involved with and does not affect the current scenario of the car's accessing the PAL entry point. Thus, there is some possibility that the current ODM might have omitted this operation during object-oriented development of the CPS system. In this situation, because of the omission of this operation, the traffic light is still in the state of "TL_GREEN" after the current car enters into the PAL; accordingly, as a negative result of this omission, another car would be incorrectly permitted to enter the PAL while the current car is still accessing the PAL. This violates the special parking assess

safety rule in the CPS system: “one access at a time”, which is certainly required to be tested in the TUC1 test scenario as part of the CPS testing requirements. But the omission of operation `setRed`() could also lead to the negative testing-related effect that we cannot exercise and examine this operation to verify whether the traffic light is in the correct state of “TL_RED” in TUC1, because this testing-required operation is omitted mistakenly and is not included as the basic test artefact for this specific testing purpose.

Therefore, if this testing-related operation is omitted in the ODM, we must abide by the CPS testing requirements to design and add it (as a supplementary basic test artefact) to the TUC1 test scenario for constructing the DOTM, and place it in the associated test sequence just after the current car enters the PAL. Just as shown in Figure 5.4, test operation `setRed`() is added after operation `clear`() in TUC1. If test operation `clear`() functions correctly (i.e. the in-PhotoCell sensor device monitors and detects that the current car properly crosses over and passes through the entry point in the PAL), the current car will have entered the PAL successfully. And then the added test operation `setRed`() must be executed in TUC1 to prevent the other car from incorrectly accessing the PAL at the same time while the current car is still accessing the PAL. This testing example has shown that it is not feasible to exercise and examine certain testing-required, *but omitted*, component/model artefacts without such supplementary testing-related artefacts added as basic test artefacts, and that accordingly this resulting “test gap” (i.e. *Test-Gap #1* as described in Section 5.2.4.2 (1)) can be bridged properly with the MBSCT methodology (especially the TCR strategy).

The above illustrative examples show that test models must contain adequate basic test artefacts, including the core existing testing-related component/model artefacts (which are identified and extracted from the existing SCD models) and the supplementary testing-related component/model artefacts (which are designed and constructed as required to add to the corresponding SCT models). The CPS DOTM construction presented in this section has demonstrated how the MBSCT methodology (especially the TCR strategy) is applied to develop adequate basic test artefacts for test model construction. Thus, the MBSCT methodology is capable of achieving an adequate set of basic test artefacts and bridging Test-Gap #1 for UML-based SCT.

5.5.3 Test Contracts for Test Model Construction

This section further discusses that appropriate complementary testing-support artefacts (e.g. test contracts as special test artefacts) are not only required for test model construction, but also very effective to bridge Test-Gap #2 and realise model-based component testability improvement for effective UML-based SCT. We apply the TbC technique and illustrate how test contracts are

designed and constructed particularly for developing the CPS DOTM.

As described in Section 5.3, built on its counterpart ODM, the DOTM mainly contains the basic test artefacts and special test artefacts. For the basic test artefacts, a basic test operation essentially exercises what the operation under test does. However, simply executing the operation under test does not always ensure that its testing is carried out properly and the relevant target testing requirement is attained. There is an important *testability* issue related to the nature of the ordinary ODM for component design and implementation: although the operation under test is exercised, it could be *not self-testable*, i.e. its testing may not be properly completed merely based on the artefacts included in the current ODM. This can occur because such a testing requirement may not be considered as part of component design and implementation with the ODM, and thus we cannot verify whether the operation under test is performed correctly with the ODM. For example, suppose the ODM includes operation `setRed`() during object-oriented development of the CPS system. However, this operation could be not self-testable simply based on the current DOM. This occurs especially when the current ODM does not contain appropriate testing-support artefacts for testing purposes (e.g. evaluating related test results). In this situation, we cannot verify whether operation `setRed`() is correctly implemented (e.g. this operation sets the traffic light to the correct state of "`TL_RED`"), and/or this operation is executed with its correct invocation for certain object interactions. Consequently, due to such *inadequate testability*, we cannot evaluate whether this operation functions correctly for the target testing requirement, even though it is included with the current ODM (or it is added to the DOTM under construction, as described above in Section 5.5.2) and it is exercised with the DOTM.

To cope with this "test gap" (i.e. *Test-Gap #2* as described in Section 5.2.4.2 (2)) for realising model-based component testability improvement, we need to design and construct appropriate complementary testing-support artefacts for the DOTM under construction, and transform and enhance the non-testable operations to be testable for effective UML-based SCT. With the TbC technique, special test contracts are developed as the complementary testing-support artefacts to verify whether the operation under test performs correctly, and to examine whether the operation integrated in the SCI context fulfils the associated object interactions and collaborations for the CIT purpose. Test contracts are typically realised and represented with testable assertions, which can be used to design *test oracles* for evaluating test results. Test contracts are constructed as special test operations added to relevant test classes for enhancing the DOTM under construction.

For the above testing example in the CPS system, we must abide by the CPS target testing requirement for operation `setRed`(), and design and construct test contract `checkState`(`trafficLight`, "`TL_RED`"), which is added as a *postcondition assertion* to verify whether

operation `setRed`() performs correctly in the TUC1 test scenario (as shown in Figure 5.4). This verification can be now carried out properly, because of testability improvement: this test contract also provides the special test state of "`TL_RED`", enabling the contract-based postcondition assertion to become *verifiable* for evaluating the expected test result, i.e. the traffic light is in the correct state of "`TL_RED`" after this operation is executed. In addition, this added test contract can be used to examine the related test message: verifying whether this operation correctly realises the associated object interaction between control class `CarController` in the car control component and device class `TrafficLight` in the device control component. Such object interaction is performed in TUC1, where control class `CarController` invokes operation `setRed`() in device class `TrafficLight` for the functional collaboration over the two control components in the CPS system. This testing example has demonstrated well that it is not achievable to examine and evaluate those testing-related, *but non-testable*, operations without such complementary testing-support artefacts (e.g. special test contracts, which are added as required), and that thus this resulting "test gap" (i.e. *Test-Gap #2* as described in Section 5.2.4.2 (2)) can be bridged properly with the MBSCT methodology (especially the TbC technique).

In addition, the TbC technique also provides a set of useful contract-based test concepts and test contract criteria to guide test contract design (which will be further discussed in Chapter 6). To test the CPS system that is required to be secure and reliable for providing high quality public access services, we apply the TbC test contract criteria for a high-level coverage of adequate test contracts (including ITCs and ETCs, which were introduced in Section 4.3.3 and will be further described in Chapter 6). With the CPS TUC1 test scenario for constructing the DOTM, we design and apply appropriate test contracts to each of the associated operations of the traffic light and in-PhotoCell sensor devices, which jointly control the test car's access to the parking entry point. As described above, a special test contract is developed to verify whether the operation under test performs correctly, and examine whether the operation-related object interactions are fulfilled correctly for the CIT purpose. Such a test contract is added to the relevant test scenario (e.g. the CPS TUC1 test scenario), and is annotated with the appropriate prefix ITC or ETC to show its property of internal or external *effectual contract scope* (this contract-based concept will be also formally defined in Chapter 6). Test contracts are also numbered with their corresponding test operations and illustrated with the shaded narrow rectangles as shown in Figure 5.4.

The above illustrative examples show that test models must contain adequate special test artefacts that are test contracts designed as verifiable testing-support artefacts, enabling the testing-related, *but non-testable*, component/model artefacts to become testable as required. The

CPS DOTM construction presented in this section has demonstrated how the MBSCT methodology (particularly the TCR strategy and the TbC technique) is applied to develop adequate special test contracts for test model construction. Therefore, the MBSCT methodology is capable of achieving adequate special test contracts, bridging Test-Gap #2 and realising model-based component testability improvement for effective UML-based SCT.

5.6 Summary and Discussion

This chapter has applied the MBSCT methodology to develop a set of UML-based test models in the first phase of the MBSCT framework. The first four MBSCT methodological components were applied to test model construction, which is UML-based, process-based, scenario-based and contract-based. The model-based integrated SCT process guides what types of test models need to be built (e.g. use case and object test models) and what relevant UML-based software models are needed as the basis for developing a particular test model.

The construction of a specific test model was undertaken in the following technical process, where the TCR strategy plays the major technical role in collaboration with the relevant MBSCT techniques:

(1) We applied the TCR strategy for test-centric model refinement to identify and extract the core set of basic test artefacts (which are testing-related component/model artefacts that are testing-required or are testable) for creating the prototype of the test model and ensuring that the test model under construction does not contain other testing-irrelevant information.

(2) We applied the TCR strategy for model-based testability improvement to design and construct appropriate supplementary testing-related component/model artefacts as the additional basic test artefacts for the test model under construction, so that they can enable certain testing-required, *but omitted*, component/model artefacts to be tested (i.e. bridging Test-Gap #1).

(3) Further applying the TCR strategy for model-based testability improvement with the TbC technique, we designed and constructed appropriate test contracts, which are used as the special test artefacts for enhancing the test model under construction, and are complementary testing-support artefacts to enable the testing-related, *but non-testable*, component/model artefacts to become testable as required (i.e. bridging Test-Gap #2).

(4) Finally, we applied the TCR strategy for test-centric model optimisation with the sce-

nario-based CIT technique to improve and optimise test model construction: we focused test model construction on developing the basic test artefacts and special test artefacts related to the core test scenarios that have high testing priority for the CIT purpose. Test models must contain adequate basic test artefacts and special test artefacts, but test models do not need to, and should not, include other redundant testing-irrelevant artefacts.

In this chapter, we have applied the MBSCT methodology to construct relevant use case and object test models for the CPS case study. The testing examples selected from the CPS case study have illustrated how the MBSCT methodology was applied to develop both adequate basic test artefacts and adequate special test contracts for test model construction. The construction of the CPS test models has well shown that the MBSCT methodology is capable of bridging the identified "test gaps" (both Test-Gap #1 and Test-Gap #2) and improving model-based component testability for effective test model construction. Therefore, this chapter has demonstrated the MBSCT testing applicability and capabilities particularly for test model construction, adequate test artefact coverage and component testability improvement (which are the core MBSCT testing capabilities #1, #4 and #5 as described earlier in Section 4.6). A more comprehensive validation and evaluation of the MBSCT methodology will be presented in Chapter 9.

A major purpose of the first phase of the MBSCT framework aimed to develop useful model-based test artefacts and construct test models as the principal foundation for UML-based SCT. The subsequent testing activities with the MBSCT framework are component test design and evaluation, which will be discussed from Chapter 6 onwards. Furthering the TbC's introduction and application to test model construction presented in Chapter 4 and Chapter 5, Chapter 6 will formally describe the TbC technique and associated technical aspects in more detail, and undertake contract-based test design for UML-based SCT.

Chapter 6
Test by Contract for UML-Based SCT

6.1 Introduction

After test model development (as described in Chapter 4), the second phase of the MBSCT framework starts with component test design to develop component test cases for component test evaluation. With the MBSCT methodology, component test development is *model-based*, which means that component tests are designed based on the constructed UML-based test models. Component test development is *process-based*, which means that the integrated SCT process guides the iterative and incremental development of test models and model-based component tests. Component test development is also *scenario-based*, which means that component tests are designed based on test scenarios for testing crucial component functional scenarios. Moreover, component test development is *contract-based*, which means that the Test by Contract (TbC) technique plays a major role in contract-based component test design. The TbC technique is one of the most important MBSCT methodological components. Chapter 4 presented a basic introduction to the TbC technique (in Section 4.3.3), and Chapter 5 applied the TbC technique to test model construction (especially in Sections 5.2.3, 5.2.4.2, 5.3, 5.4.3 and 5.5.3). This chapter formally describes the TbC technique and related technical aspects in more detail [173] [175] [176].

The TbC technique is introduced for the principal goal of bridging Test-Gap #2 and improving component testability in model-based component testing. In Section 2.6, we have studied the component testability concept and characteristics, and reviewed the main strategies and approaches for component testability improvement. Technically, these approaches (especially the first three approaches as described earlier in Section 2.6.2) are in line with the general idea of assertions [164] [151] [152] [123] [153] and the Design by Contract (DbC) concept [91] [92]. However, they mainly employ a traditional approach to inserting some test artefacts (e.g. assertions) inside component programs at the level of source code. Such a traditional approach may be applicable to code-based testing, but has certain limitations to effectively support model-based approaches to component integration testing (CIT) at the model-based specification level. This research takes a different approach to overcome those limitations by incorporating appropriate testing-support artefacts (e.g. special test contracts) at the model-based specification level to bridge Test-Gap #2 and improve model-based component testability with model-based test contracts. This ensures that model-based testability stands at a test level above traditional code-based testability and thus effectively supports model-based approaches to SCT.

The DbC concept was originally proposed by Meyer in designing traditional software classes, and was used to formalise the contract relationship between a supplier class and its clients, and define the associated object-oriented design elements. While it might not be initially considered as a SCT technique, the DbC concept supports the common testing goal of assuring component correctness and quality. This research adapts the idea of the DbC concept and applies it to bridge Test-Gap #2 and improve software component testability particularly for UML-based CIT. We investigate the following key testing-related questions:

(1) Can the DbC concept be combined with UML models for UML-based CIT, beyond the DbC's initial object-oriented class level?

(2) How can the DbC concept be used to improve component testability for CIT? In particular, this issue has two further associated aspects as follows:

(a) How can the DbC concept be adapted and then applied to facilitate component test design and generation?

(b) How can the DbC concept be adapted and then applied to facilitate component fault detection and diagnosis?

(3) Can the DbC concept be further extended to develop a new contract-based approach for CIT with UML models?

We argue that the combination of UML-based testing and the DbC concept is an effective approach for bridging the "test gaps" in UML-based testing and improving model-based component testability for effective UML-based SCT. The TbC technique is introduced as a new contract-based SCT technique to address these important testing issues.

This chapter formally describes the TbC technique. Section 6.2 presents a technical overview of the TbC technique and describes a stepwise TbC working process. Section 6.3 discusses the TbC foundation principles to support the primary goal of *Contract for Testability*. This is accompanied with a set of important contract-based test concepts and associated technical aspects we have developed for the TbC technique. In particular, Section 6.3.1 formally introduces the *test contract concept*. Section 6.3.2 discusses how to realise and represent test contracts for test contract design. Section 6.3.3 introduces the *effectual contract scope* concept, and describes different categories of *internal/external test contracts* and their testing relationships. Section 6.3.4 introduces a set of new *TbC test contract criteria* and discusses how they are used for contract-based SCT. Section 6.3.5 describes how the TbC technique can improve component testability characteristics. Then, we move on to applying the TbC technique to UML-based SCT, and employ the CPS case study to illustrate by examples how to put the TbC technique into

practice to undertake contract-based SCT with UML models. Section 6.4 applies the TbC technique to undertake test contract design for test model construction. Section 6.5 discusses contract-based component test design. Section 6.6 discusses related work and describes the main characteristics of the TbC technique. Section 6.7 presents our summary of this chapter.

6.2 Test by Contract: An Overview

The *Test by Contract* (TbC) technique is developed to be a new contract-based SCT technique that extends the DbC concept to the new domain for UML-based SCT, beyond the original DbC scope for code-based unit testing of traditional software classes. By introducing the primary concept of a *test contract* (TC), we further develop a set of useful contract-based test concepts and test contract criteria, which establish the technical foundation for the TbC technique. On this basis, the TbC technique employs the useful testing-support mechanism of *test contracts*, and designs and constructs appropriate test contracts to undertake contract-based SCT activities.

Figure 6.1 illustrates a typical stepwise TbC working process with five major TbC steps to carry out key testing tasks with the TbC technique. This stepwise testing process shows how to put the TbC technique into practice for contract-based testing activities to undertake UML-based SCT, which is summarised as follows:

(1) Step TbC1 deals with the test contract concept, and basic characteristics of test contracts (see Section 6.3.1 to Section 6.3.3);
(2) Step TbC2 deals with test contract design to improve model-based testability and enhance test model construction (see Section 6.3.4 to Section 6.3.5, and Section 6.4);
(3) Step TbC3 deals with contract-based test design based on test models (see Section 6.3 to Section 6.5);
(4) Step TbC4 deals with fault detection and diagnosis with contract-based test design, which aims to achieve the goal of effective component test design (see Chapter 7). Also contract-based fault detection and diagnosis in Step TbC4 is a central part of component test evaluation (see Chapter 9);
(5) Step TbC5 deals with contract-based test generation (see Chapter 8).

Technically, the overall TbC working process comprises two main phases: Steps TbC1, TbC2 and TbC3 form the TbC *foundation phase*, and Steps TbC3, TbC4, and TbC5 form the TbC *advanced phase* (as shown in Figure 6.1). In particular, Step TbC3 is the kernel of the TbC technique, which is based on test contract design with test models and aims to undertake contract-based test design to detect and diagnose component faults and to generate contract-based component tests.

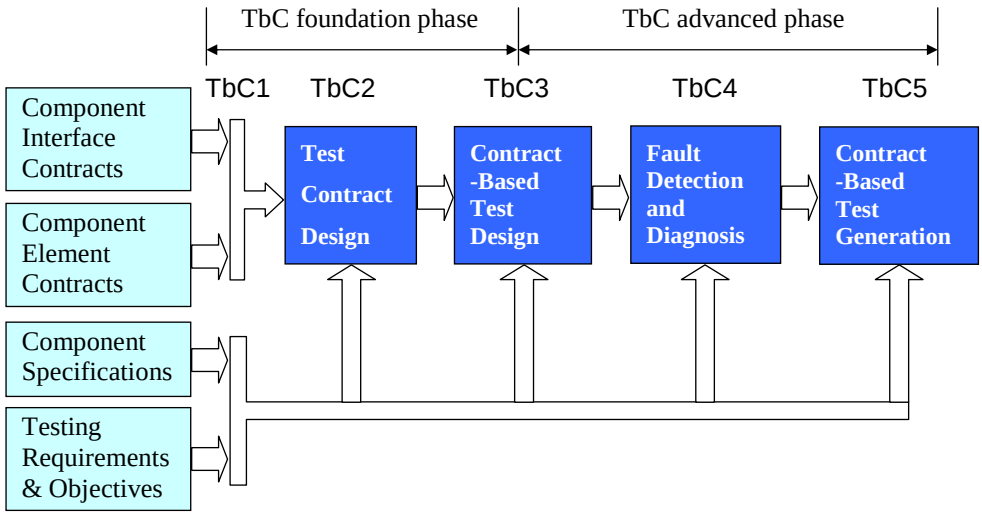


Figure 6.1 Test by Contract: Stepwise TbC Working Process

6.3 Contract for Testability

The TbC technique is a goal-driven SCT approach to achieve the central testing goals of the *Contract for Testability* (CfT) concept, which aims to:

(a) design and construct appropriate test contracts for bridging the "*test gaps*" and improving component testability in UML-based SCT;
(b) apply and supplement test contracts for developing testable components;
(c) conduct and facilitate component test design and generation for conformance to target testing requirements;
(d) detect and diagnose component faults for achieving target testing objectives;
(e) evaluate and demonstrate the required level of component correctness and quality.

The above CfT goals actually involve two major parts: (1) *testability specification and improvement* (covering CfT goals (a) – (c)), which are the primary CfT goals, and are mainly discussed in this chapter and Chapter 8; (2) *testability verification and evaluation* (covering CfT goals (d) – (e)), which are the higher-level CfT goals, and will be discussed in Chapter 7 and Chapter 9. The two CfT parts work collaboratively together to achieve effective SCT.

To support the stepwise TbC working process and the CfT goals, we develop a set of important contract-based test concepts, test contract criteria and associated technical aspects. This section describes these essential TbC foundation aspects related to Steps TbC1, TbC2 and TbC3 of the TbC foundation phase (as shown in Figure 6.1). This extends the basic introduction to the TbC technique as described earlier in Section 4.3.3, and further discusses the TbC technique in more detail. The later sections of this chapter (see Section 6.4 to Section 6.5) employ the CPS

case study to illustrate by examples the relevant contract-based test concepts and test contract criteria, and how they are applied to contract-based test model construction and component test design.

6.3.1 Test Contract Concept

One key feature of the TbC technique is that it clearly focuses on identifying and designing what we call test contracts, which is the primary testing-support mechanism to achieve the CfT goals. For software component integration (SCI), both component developers and users reuse and deploy a particular component as an encapsulated software unit mainly via its component interfaces, which define certain contractual rules between composite components in the integration context. These *component (interface) contracts* specify how to use component interfaces correctly to access component functional services (which are typically represented and realised with component operations) for SCI. In particular, component contracts capture the mutual responsibilities (e.g. obligations and benefits) that both partners of a component (i.e. service supplier/contractor and client) must comply with, independent of how they are fulfilled and implemented. Component contracts govern the operations and interactions of composite component objects that are integrated into a component framework, application or component-based system. Any occurrence of *contract violation* indicates one or more potential *component faults* resulting from incorrect component design. In addition to contracts specified at the component interface level, we can also design *component element contracts* at the component element level to examine certain low-level component elements underlying the component interface for in-depth testing coverage. In particular, component element contracts can be used to verify a specific component state or an individual underlying object operation that composes the component operation under test for the CIT purposes. From the viewpoint of component contracts, a major task of CIT is to design appropriate test contracts, develop contract-based component tests, and examine component integration to conform to the specified component contracts for the target testing objectives and requirements.

The new test contract concept introduced in the TbC technique adapts the contract notion used by [139] in defining software component interfaces. The new test contract concept extends the contract notion used with the DbC concept [91] [92] in designing software classes to the new domain of UML-based SCT (as described earlier in Section 4.3.3). Based on the new test contract concept for undertaking contract-based SCT, we design and construct test contracts based on relevant component contracts (e.g. interface-level contracts, element-level contracts, etc.), and test contracts work as the primary testing-support mechanism to improve component testability and support the CfT goals. This indicates that the TbC technique also supports test-driven development particularly for contract-based testing.

6.3.2 Realising and Representing Test Contracts

This subsection describes software realisation and representation of test contracts, which provide the basis for test contract design (Step TbC2) towards contract-based test design and generation (Steps TbC3 and TbC5). Technically, test contracts can be applied to various testing-related artefacts at different modeling levels in test model construction (see Section 6.4) and at different test levels/phases (e.g. integration/unit testing, see Section 6.3.3) for the CfT goals. In practice, test contracts are usually realised and represented with *assertions* and associated concepts in the form of commonly-used *preconditions*, *postconditions* and *invariants* [24] to design contract-based component tests. Table 6.1 summarises the main forms of test contracts and their relationships with the main model and component artefacts.

Table 6.1 Test by Contract: Model/Component Artefact, Contract Artefact

Model	Model/Component Artefact	Contract Artefact
Use case model	Use case	Pre/postcondition, invariant
	Scenario	Pre/postcondition, invariant
	Sequence	Pre/postcondition
	System behaviour	Pre/postcondition
	System operation/event	Pre/postcondition
	System state	Pre/postcondition, invariant
Object model	Scenario	Pre/postcondition, invariant
	Sequence	Pre/postcondition
	Behaviour	Pre/postcondition
	Message	Pre/postcondition
	Operation/event	Pre/postcondition
	Class	Pre/postcondition, invariant
	(object) state (attribute)	Pre/postcondition, invariant

Conceptually, an *assertion* is a formal constraint or condition that describes certain semantic properties of software artefacts. An assertion is expressed as a logical *Boolean predicate* whose value is either *true* or *false* when it is evaluated. An assertion is *verifiable* or *testable*: *true* indicates that the software artefact concerned conforms to the required software property; *false* indicates an error or fault, which means that the software artefact concerned violates the required software property. Among the three common types of assertions, preconditions and postconditions form the basic assertions that can be applied to almost all component/model artefacts, and invariants are usually applied to classes, scenarios and use cases, as well as states of components/objects. For example, for an operation under test, a *precondition* is an assertion de-

fining certain properties that must hold *true* before the operation is invoked and executed. A *postcondition* is an assertion defining certain properties that must hold *true* after the completion of the operation's execution. An *invariant* is an assertion defining certain properties that must hold *true* at all times in the scope of a class, scenario or use case, or for a state of a component/object.

From Table 6.1, we can see that test contracts can be applied in different forms in different software contexts. In the context of component artefacts, a test contract for an operation is composed of basic assertions (preconditions and/or postconditions) that are applied and evaluated before and/or after the execution of the operation. In addition to the basic assertions, a test contract for a component class unit may be a class invariant (if applicable). In the context of model artefacts for model-based CIT, a test contract in a test model is a special test message/operation that aims to verify relevant collaboration messages/operations between interacting objects in the SCI context. A special test message/operation, which behaves in the way being consistent with an ordinary message/operation for testing purposes, will be mapped and transformed to one or more concrete test operations that are finally realised with appropriate assertions for testing component artefacts (see the earlier Section 4.3.5 for the CTM technique). The TbC technique uses special test operations to represent test contracts, which are composed of common assertions for verifying component artefacts. They are developed to be compatible with the usual operations of components or classes (but such special assertion-based test operations should have one of two possible Boolean return values, *true* or *false*), and thus are able to be executable with component programs to support dynamic testing.

In the same manner as common assertions, test contracts represented with assertions should be side-effect free (see Section 6.5.2 for relevant illustrative examples), and should not affect or change the important sequencing attribute of related test sequences (see Section 6.5.1 for test sequence design and relevant illustrative examples), when they are used as special testing-support artefacts to improve component testability and facilitate component test design. Based on the feature of assertions being verifiable or testable, test contracts represented with testable assertions can be used as the basis to design relevant *test oracles* for verifying test cases and evaluating test results. These characteristics are very important for test contract design and contract-based test design (which is to be further discussed in Section 6.5).

6.3.3 Effectual Contract Scope – Internal/External Test Contract

This section further explores some important contract-based test concepts and characteristics. First, we introduce the effectual contract scope concept, and describe different categories of internal/external test contracts. Then, we discuss the relationships between internal and external test contracts, and the relationship between internal/external test contracts and test levels.

6.3.3.1 Effectual Contract Scope

Any software artefact (e.g. an ordinary class attribute or operation) has an existence context with a scope of access and visibility. To deal with test contracts effectively, we introduce an important new concept for a test contract: *effectual contract scope*, which refers to a software context (e.g. a component context or model context) in which the test contract can take effect (e.g. the test contract can be verified for a particular testing purpose). A test contract functions relative to its effectual contract scope. The importance of this concept is that it indicates how a particular test contract actually affects the extent and outcome of the required testing that is related to this test contract.

6.3.3.2 Categories of Test Contracts

Based on the effectual contract scope concept, we can explore the relationship between the effectual contract scope and the software context of a test contract, and classify test contracts into two main categories (as shown in Figure 6.2):

(1) An *internal test contract* (ITC) is defined and applied to, and is also verified within, the same effectual contract scope and the same software context, i.e. both have the same component/model context. For example, a test contract for a class attribute (object state) is normally an ITC (as shown in Figure 6.2).

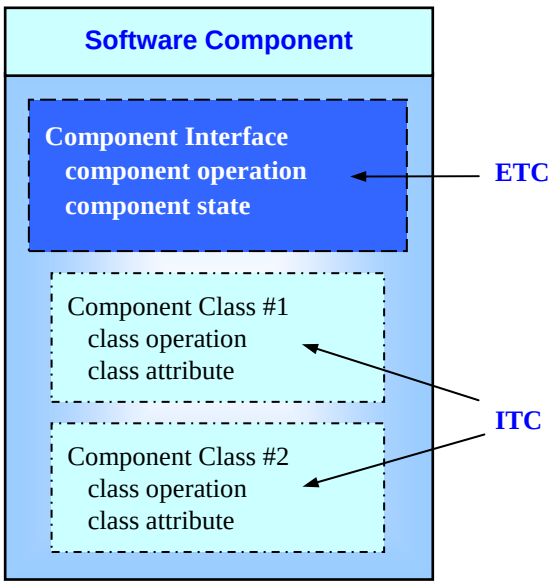


Figure 6.2 Test Contracts: ITC and ETC

(2) An *external test contract* (ETC) is defined and applied to a software context, but is verified outside this software context. This indicates that the effectual contract scope of the ETC is not the same as its software context. For example, a test contract for a component operation is usually an ETC (as shown in Figure 6.2).

6.3.3.3 Relationships between Internal and External Test Contracts

Whether a test contract is internal or external really depends on its effectual contract scope, and this characteristic is usually not subject to where the test contract is defined. In some situations, the type of a test contract may turn into another type when the extent of its effectual contract scope is changed. To illustrate this, let us examine the following situations:

(i) If the scope narrows, an ITC in the original effectual contract scope may become an ETC outside the new narrowed scope.

For example, an ITC of a component may become an ETC of a constituent class in this component, when this test contract becomes conceptually external to this class. Such refinement of the effectual contract scope is necessary and useful to clarify the actual relationship of the test contract to its host class within the component.

(ii) If the scope broadens, an ETC in the original effectual contract scope may become an ITC inside the new broadened scope.

For example, an ETC of a class may become an ITC of a new component, when this class becomes part of this new component. While this scope shifting is appropriate, it is especially useful to identify and construct the proper type of test contracts (ITCs and ETCs) for this new component, when we develop components made up of different classes.

It is very important to analyse and recognise the properties of these types of test contracts and their relationships, in order to design and apply appropriate types of test contracts for contract-based testing:

(a) Usually, an ITC exists independent of any ETCs. Verifying an ITC is usually irrelevant to the verification of any ETCs. However, an ITC may provide some testing-support artefacts for one or more related ETCs.

(b) By contrast, an ETC may relate to, or depend on, one or more ITCs, where an ETC may be composed of some ITCs and other test artefacts, and verifying this ETC may require the verification of the associated, underlying ITCs.

6.3.3.4 Test Contracts and Test Levels

In principle, test contracts are applicable to various software artefacts at different test levels/phases for conducting SCT. In practice, ITCs and ETCs can work at their particular test levels, which are slightly different but are still relevant, as illustrated by the following points:

(1) ITCs are often used in unit testing of the component, but they are required to be re-examined in the CIT context where they are used.
(2) By contrast, ETCs are often used in CIT, where an ETC is verified in one integration module (e.g. an integration class that controls several underlying integrated classes) whereas this ETC is defined and applied to another integration module. When an ETC includes some underlying constituent ITCs in the effectual contract scope, these associated ITCs are required to be verified along with this ETC.
(3) ITCs are often used to trace and examine internal component/object states, e.g. for the purpose of unit testing.
(4) ETCs are typically used to trace and examine external component operations/events and states, e.g. for the CIT purposes.
(5) In the same manner as common assertions, ITCs and ETCs should be side-effect free when they are used to examine and trace relevant testing information (the relevant illustrative examples will be provided in Section 6.5.2).

6.3.4 Contract-Based Test Criteria

To support the CfT goals, we need to develop a contract-based testing guide for test contract design with effective measurable test contract coverage and adequacy rules or requirements. This section discusses the development of useful contract-based test criteria for the TbC technique, which are called the *TbC test contract criteria*.

6.3.4.1 Setting TbC Test Contract Criteria

In principle, test criteria refer to the criteria that a system or component must meet in order to pass a given test [77]. Test criteria are regarded as very useful testing guidelines, rules or requirements to enhance and thus ensure testing quality. For the TbC technique, we study test criteria in the context of test contracts, and seek a set of useful contract-based test criteria to guide how to design and apply adequate test contracts effectively for achieving the CfT goals. The TbC test contract criteria are developed to support test contract design, contract-based test design and fault detection and diagnosis (as described in Steps TbC2, TbC3 and TbC4 shown in the stepwise TbC working process in Figure 6.1). Technically, we mainly focus on two crucial

aspects of the TbC test contract criteria: test contract coverage and test contract adequacy:

(a) TbC Test Contract Criteria: test contract coverage

Test contract coverage refers to the *extent* to which one test contract or a set of test contracts can properly exercise and examine the specified test requirement for a given component artefact, component or system under test. Good test contract coverage criteria require appropriate test contracts to *cover and examine* each important component artefact that is required to be tested, according to all the specified test requirements for testing of the entire component or component-based system under test.

(b) TbC Test Contract Criteria: test contract adequacy

Test contract adequacy refers to the *quality* of one or more test contracts that are able to *sufficiently* meet a specified testing requirement correctly and satisfactorily. Good test contract adequacy criteria require that a certain *minimal amount* of appropriate and necessary test contracts can *sufficiently cover and examine* each of the important component artefacts that are required to be tested, in order to comply with all the specified testing requirements correctly and satisfactorily.

A major purpose of the TbC test contract criteria is to provide practical testing guidelines for test contract design and construction to support the CfT goals. The TbC test contract criteria for test contract coverage aim to guide what test contracts are needed for effective test design to cover and examine possible component artefacts to improve component testability. Because there are various levels of granularity of component software composition and formation (as described earlier in Section 2.2.3), we need to design and construct adequate test contracts to exercise and examine different types of component artefacts at different complexity levels. The TbC test contract criteria are created to accommodate important testing-related component artefacts under test, such as states, events, operations, classes and components, which are all the essential software constituents to compose and construct final executable programs of software components and systems under test. For the practical, achievable testing purpose, we base the "*adequacy*" of test contracts on the testing-required component artefacts that are sufficiently covered by appropriate test contracts for the goal of desired test effectiveness.

We introduce a set of new TbC test contract criteria for adequate test contract coverage shown in Table 6.2, in order to provide practical testing guidelines for test contract design to support the CfT goals. They comprise a collection of contract-based testing rules that impose certain mandatory testing requirements on a set of relevant test contracts to adequately cover and examine the important testing-related component artefacts for effective test design. All the TbC test contract criteria #1 to #6 shown in Table 6.2 provide structural coverage measures and

can be categorised into three different levels. The low-level TbC test contract criteria #1 and #2 focus on component elements and form a foundation for other TbC test contract criteria. As the middle-level test contract criteria, TbC test contract criteria #3 and #4 work on the component unit level. Test contracts for the high-level TbC test contract criteria #5 and #6 focus on the overall component level and are usually composed of certain relevant test contracts used for the underlying lower-level TbC test contract criteria. In this case, verifying a test contract for a higher-level TbC test contract criterion (e.g. TbC test contract criterion #5 or #6) requires the examination of all constituent test contracts used for the lower-level TbC test contract criteria. As described in Section 6.3.2, test contracts represented with basic assertions (preconditions and postconditions) can be applied to all the TbC test contract criteria #1 to #6 shown in Table 6.2. Test contracts represented with invariant assertions are usually applicable to the TbC test contract criteria #1, #4 and #6, if the associated component artefact has an invariant property.

Table 6.2 Test by Contract: TbC Test Contract Criteria

	No.	Test Criterion	Description
Low-Level Test Criteria	#1	Test state coverage criterion	The test contract set must contain adequate test contracts that can test and check each state of the component or its objects under test.
	#2	Test event coverage criterion	The test contract set must contain adequate test contracts that can test and examine each event pertinent to the component or its objects under test.
Middle-Level Test Criteria	#3	Class-operation-level test contract coverage criterion	The test contract set must contain adequate test contracts that can test and check each constitute (public) class operation that contributes to the (full or partial) formation of a component operation under test.
	#4	Component-unit-level test contract coverage criterion	The test contract set must contain adequate test contracts that can test and examine each constituent class unit that contributes to the (full or partial) formation of the component under test.
High-Level Test Criteria	#5	Component-operation-level test contract coverage criterion	The test contract set must contain adequate test contracts that can test and check each operation of the component under test.
	#6	Component-level test contract coverage criterion	The test contract set must contain adequate test contracts that can test and examine the component under test.

The description of the TbC test contract criteria shown in Table 6.2 focuses on the component under test (CUT) as the major subject of SCT. However, the TbC test contract criteria we develop are also applicable to similar software modules, such as individual classes/objects with well-defined interfaces. The following subsections further discuss each of the TbC test contract criteria in detail, especially how they are used and their relationships for contract-based SCT.

6.3.4.2 TbC Test Contract Criterion #1: test state coverage criterion

TbC Test Contract Criterion #1: test state coverage criterion

The test contract set must contain adequate test contracts that can test and check each state of the component or its objects under test.

Component states capture certain useful testing information about component existence conditions, attributes, properties and/or relationships with other peer components/objects in time. Components may reside in multiple states at any one time. A component must satisfy its related state conditions or constraints for the software correctness purpose. A state invariant indicates that the component must have certain consistently-required conditions in a specified environment for a specified time. Test contracts for this TbC test contract criterion may be used as part of test contracts at the class unit level (see TbC test contract criterion #4) and the component level (see TbC test contract criterion #6) as well as some other test contracts if applicable. In this case, verifying test contracts at the class level or component unit level requires the verification of test contracts for checking underlying associated states.

In addition, test states may be also associated with some related test events that may affect the state's attributes, conditions and/or termination. In this case, test contracts for checking test states are associated with test contracts for checking related test events (see TbC test contract criterion #2 below).

6.3.4.3 TbC Test Contract Criterion #2: test event coverage criterion

TbC Test Contract Criterion #2: test event coverage criterion

The test contract set must contain adequate test contracts that can test and examine each event pertinent to the component or its objects under test.

An event is associated with a relevant occurrence of message sending (e.g. object communication for collaboration), response reception (e.g. from a server, class or component), state transition stimulus (e.g. in a state machine), or external service request (e.g. from the user in a GUI context). A fired or triggered event can activate the execution of an event operation, which may change certain associated states of the CUT. Test contracts for this TbC test contract criterion may be used as part of test contracts at the class operation level (see TbC test contract criterion #3) and component operation level (see TbC test contract criterion #5) as well as some other test contracts if applicable. In this case, verifying test contracts at the class operation level or component operation level requires the verification of test contracts for checking associated events.

In addition, test contracts for checking test events may be also associated with test contracts for checking certain associated test states affected by the event's occurrence and execution. In this case, the examination of the test contracts for checking test events leads to the verification of the test contracts for checking the event-associated test states (see TbC test contract criterion #1 above).

6.3.4.4 TbC Test Contract Criterion #3: class-operation-level test contract coverage criterion

TbC Test Contract Criterion #3: class-operation-level test contract coverage criterion

The test contract set must contain adequate test contracts that can test and check each constitute (public) class operation that contributes to the (full or partial) formation of a component operation under test.

A component operation is typically realised with one or more class operations from one or more underlying class units, which constitute the component where this component operation exists. Public operations of class units are typical candidates for constructing component operations, which are a key basis for component interface design. This TbC test contract criterion ensures that necessary test contracts can cover and examine important class operations, which establishes a coverage basis for other TbC test contract criteria covering component units (see TbC test contract criterion #4) and component operations (see TbC test contract criterion #5).

6.3.4.5 TbC Test Contract Criterion #4: component-unit-level test contract coverage criterion

TbC Test Contract Criterion #4: component-unit-level test contract coverage criterion

The test contract set must contain adequate test contracts that can test and examine each constituent class unit that contributes to the (full or partial) formation of the component under test.

A class is regarded as the basic software unit composing a software component. This TbC test contract criterion requires necessary test contracts to cover and examine certain underlying component artefacts inside the component. For this TbC test contract criterion, test contracts can be (fully or partially) composed of test contracts at the class operation level (as described in TbC test contract criterion #3), and test contracts covering component states/events in the class unit (see TbC test contract criterion #1 and #2), as well as some additional test contracts as necessary. In this case, verifying a test contract at the component unit level requires the verification

of all underlying test contracts related to testing-related class operations and elements in the component unit. Note that, whether or not a component class has any invariant properties depends on the actual component requirements and specifications. Accordingly, this test contract coverage may not always include assertions for class invariants.

6.3.4.6 TbC Test Contract Criterion #5: component-operation-level test contract coverage criterion

TbC Test Contract Criterion #5: Component-operation-level test contract coverage criterion

The test contract set must contain adequate test contracts that can test and check each operation of the component under test.

Component operations are specified mainly through the well-defined component interface, which is used as the basic means for accessing component functions. Because a component operation typically consists of several class operations from the component's underlying composite classes, test contracts for this TbC test contract criterion can be (fully or partially) composed of test contracts used at the class operation level (as described in TbC test contract criterion #3) and some additional test contracts as necessary. In this case, verifying a test contract at the component operation level requires the verification of all constituent test contracts at the class operation level.

Moreover, component functional testing mainly examines the component interface and undertakes component operation testing. Following this TbC test contract criterion, applying adequate test contracts to cover and examine all component operations can effectively support component interface testing and thus component functional testing.

6.3.4.7 TbC Test Contract Criterion #6: component-level test contract coverage criterion

TbC Test Contract Criterion #6: Component-level test contract coverage criterion

The test contract set must contain adequate test contracts that can test and examine the component under test.

Testing individual components is the foundation of testing component-based systems that are composed of software components. Because a component under test is usually composed of multiple underlying composite classes and component operations defined through the component interface, we actually need a set of appropriate test contracts for testing the CUT in two main aspects:

(a) For component functional testing:

This TbC test contract criterion requires sufficient test contracts to cover and examine each of the component operations specified through the component interface (as described in TbC test contract criterion #5). Accordingly, this TbC test contract criterion requires sufficient test contracts to test all component operations and the component interface. This TbC test contract criterion works based on TbC test contract criterion #5 for the purpose of component functional testing.

(b) For component structural testing of the underlying component artefacts behind the component interface:

This TbC test contract criterion requires sufficient test contracts to cover and examine each of the underlying composite classes (as described in TbC test contract criterion #4) and component elements (as described in TbC test contract criteria #1 and #2) inside the component. Accordingly, this TbC test contract criterion works based on TbC test contract criteria #4, #1 and #2 for the purpose of component structural testing.

6.3.4.8 Adequate Test Contract Coverage and Testing Efficiency

A major purpose of the TbC test contract criteria for adequate test contract coverage promotes and supports a high-level coverage of adequate test contracts that are applied to possible component operations and elements under test (e.g. for testing safety-critical software components and systems). However, the high-level coverage of adequate test contracts would attract higher testing overheads, and lead to low testing performance and efficiency. On the other hand, this would also produce the result that some testing, which requires higher-level adequate test contract coverage, could become unattainable and infeasible in testing practice, due to the increasing size and complexity of software components and systems under test.

In testing practice, the necessary extent of adequate test contract coverage really depends on the actual testing requirements and objectives. An appropriate trade-off between test contracts, testing overheads and efficiency requires that test contract coverage needed for test design should be as minimal and as adequate as possible to meet the required level of target testing requirements and objectives.

6.3.5 Realising Component Testability Characteristics Improvement

A major goal of the TbC technique is to improve component testability. As described earlier in Section 2.6.1, the first three characteristics of component testability (i.e. traceability, observability and controllability) are very important for providing good component testability. To realise component testability improvement, the TbC technique particularly employs the test contract mechanism and the TbC test contract criteria to design and apply adequate test contracts to enhance the three important component testability characteristics.

(1) *Improving component traceability*. Adequate test contracts can examine different component traces concerning component behaviours and related software elements, such as state, event, operation, etc. Because these traceable artefacts may exist internally (inside a component) or externally (on the component interface), test contracts can trace and record component execution and test execution information in both white-box and black-box views.

(2) *Improving component observability*. Based on component information traced with adequate test contracts, we can observe dynamic information of component functions, testing-related behaviours and certain possible failure information. In particular, test contracts can aid monitoring and examination of *input-output inconsistency* of component tests, which is a key property that affects component observability.

(3) *Improving component controllability*. By enhancing component traceability and observability, we are able to control the process of component execution and test verification. We can observe specific traced test information (e.g. with initial test states as test inputs) to monitor and control related test outputs (e.g. resulting test states) during testing. Such a *test-input-output correlation* is very important to evaluate the observed test results, and determine test passes or fails of test execution for assessing the expected component correctness.

6.4 Test Contract Design for Test Model Construction

After introducing the TbC foundation aspects (including the contract-oriented concepts and TbC test contract criteria), we follow the stepwise TbC working process (as shown in Figure 6.1), and use the CPS case study to illustrate how to put the TbC technique into practice particularly for undertaking contract-based CIT with UML models. One important objective is to demonstrate the applicability and effectiveness of the TbC technique for UML-based SCT. This section focuses on test contract design for contract-based test model construction (i.e. Step TbC2).

To support the CfT goals for UML-based SCT, test contract design aims to bridge the "test gaps" in UML-based SCT and improve model-based component testability for effective test model construction. For this purpose, the TbC technique works together with the relevant MBSCT methodological components, as described earlier in Chapter 5 (especially in Sections 5.2.3, 5.2.4.2, 5.3, 5.4.3 and 5.5.3). Also as described earlier in Section 5.3, we classify test artefacts used in test model construction into two main categories: basic test artefacts and special test artefacts. Our strategy for test contract design focuses on developing effective model-level

test contracts as the special test artefacts to realise component testability at the modeling level and improve test model construction. We design adequate test contracts, and apply them as complementary testing-support artefacts to enable the testing-required, but non-testable component/model artefacts (which are in the category of basic test artefacts) to become testable as required for contract-based test model construction. In particular, for testing the CPS system that is required to be secure and reliable to provide high quality public access services, we need to apply the TbC test contract criteria for a high-level coverage of adequate test contracts. In other words, we design and apply sufficient test contracts (including ITCs and ETCs) to all parking control operations of the related CPS control devices that jointly manage the car's access to the PAL. In Chapter 5, Sections 5.2.3 and 5.2.4.2 have clearly described these technical aspects of the TbC technique. Sections 5.4.3 and 5.5.3 have illustrated by examples (selected from the CPS case study) to demonstrate that test contracts designed with the TbC technique are capable of bridging the "test gaps" (especially Test-Gap #2) and improving model-based component testability for effective test model construction.

6.5 Contract-Based Component Test Design

This section focuses on the core Step TbC3 (as shown in Figure 6.1) to undertake contract-based component test design with the TbC technique for CIT. In UML-based SCT, test design is carried out based on UML-based test models that are constructed and enhanced with test contract design (i.e. Step TbC2 as described in Section 6.4). We further use some selected examples of the CPS case study to illustrate how to undertake contract-based component test design for effective CIT.

Note that we employ some naming conventions for acronyms/abbreviations of the following testing terms in the MBSCT methodology: TS – test sequence/scenario, TG – test group, TO – test operation, TC – test contract, and ITC/ETC – internal/external test contract.

6.5.1 Designing Test Sequences and Test Groups with Test Contracts

6.5.1.1 Designing Test Sequences

For the CIT purpose, an important testing focus is to test component interactions, especially verifying related underlying object interactions and object state changes with those interactions, because SCI takes place mainly with the interactions through the interfaces of component objects in the SCI context. Our contract-based component test design for CIT is based on a test model that captures a sequence of test artefacts to realise test scenarios for testing relevant functional integration scenarios. A *test sequence* (TS) refers to a sequence of logically-ordered re-

lated test artefacts, such as test operations (TOs), test elements (e.g. test state, test event), test contracts (TCs), etc. Technically, test design can start with test sequence design, and combine a set of related test operations and test contracts together into an appropriate test sequence (e.g. a *test group*, which is defined in the next Section 6.5.1.2) to verify inter-component/object interactions for CIT. This testing requires well-designed test contracts to isolate, track down and examine different component traces (including not only operations but also states and events), which are important test contracts to improve component traceability and support contract-based component test design for effective CIT.

Test sequences play the key role of organising and structuring test artefacts for effective contract-based component test design. Our test design is undertaken in conjunction with test sequence design based on test models, in which test sequences are mainly mapped and derived from related scenario-based test models. For testing the CPS system, Figure 6.3 illustrates a typical *overall test sequence* that is designed and derived from the corresponding DOTM (as shown earlier in Figure 5.4) and forms the foundation of contract-based component test design to examine the CPS TUC1 integration testing scenario. This test sequence incorporates logically-ordered relevant test contracts and test operations, and special test states and test events to conduct CIT for the CPS system. Test contracts verify relevant component/object artefacts (e.g. operation, state, or event) in the associated test sequence by using appropriate testable assertions in terms of preconditions, postconditions or invariants (as described in Section 6.3.2).

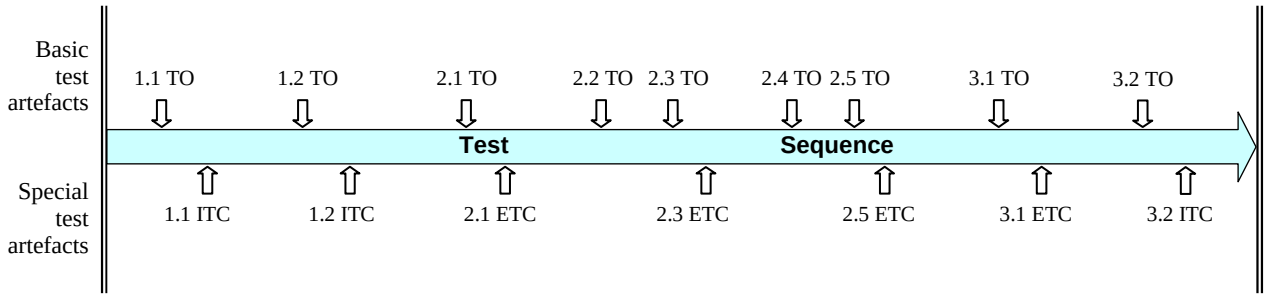


Figure 6.3 Test Sequence = test contracts + test operations (CPS TUC1 Test Scenario)

Note that when a test contract (e.g. test contract 2.5 ETC shown in Figure 6.3) is positioned between two consecutive operations under test in a test sequence (e.g. test operations 2.5 TO and 3.1 TO shown in Figure 6.3), this test contract actually plays *dual testing roles*: it works as a postcondition assertion of the last operation (e.g. test operation 2.5 TO), and also as a precondition assertion of the next operation (e.g. test operation 3.1 TO). In principle, such dual testing roles of well-designed test contracts apply to two sequential operations, if any software artefact between them does not affect the postcondition of the first operation and the precondition of the second operation. This is one of the good characteristics of the TbC technique, which demonstrates that test contracts are a useful concept for testing-support artefacts that can im-

prove component testability efficiently. Based on this TbC feature, most test contracts designed in the above CPS TUC1 test sequence functionally play such dual testing roles (as shown in Figure 6.3), which supports low-overhead test contract usage for desired testing efficiency. In addition, when a test contract is added to a sequence of test artefacts for improving testability, it does not affect or change the sequencing attribute of the test sequence (as indicated in Section 6.3.2). For example, when test contract 2.5 ETC is added to the above CPS TUC1 test sequence (as shown in Figure 6.3), test operations 2.5 TO is still verified as expected before test operation 3.1 TO and the related logical order or sequencing attribute of this test sequence remains unchanged.

6.5.1.2 Optimising Test Sequences

This subsection further explores how to optimise the structural organisation of test sequences for effective contract-based component test design. The above CPS TUC1 test sequence is designed based on the corresponding CPS TUC1 integration testing scenario, which is actually composed of three sub test scenarios (as shown earlier in Figure 5.4). Accordingly, the overall CPS TUC1 test sequence can be decomposed into three sub test sequences (as illustrated in Figure 6.4):

(1) Sub test sequence #1 examines sub test scenario #1: testing whether the stopping bar is in the expected state of "SB_DOWN" and the traffic light device is in the expected state of "TL_GREEN" (which are the CPS TUC1 preconditions as described earlier in Section 5.4.3). If so, the test car is allowed to enter and start access to the PAL.

(2) Sub test sequence #2 examines sub test scenario #2: testing whether the test car correctly enters and passes through the PAL entry point. If so, the test car has entered the PAL.

(3) Sub test sequence #3 examines sub test scenario #3: testing the traffic light device is in the expected state of "TL_RED" (which is the CPS TUC1 precondition as described earlier in Section 5.4.3). If so, the testing of the CPS TUC1 test scenario has completed.

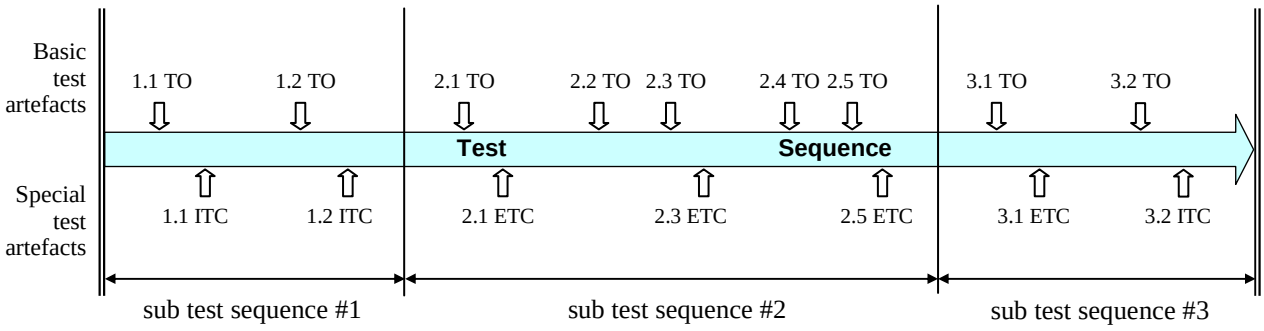


Figure 6.4 Structured Test Sequence = a series of sub test sequences (CPS TUC1 Test Scenario)

In other words, when a test scenario is logically composed of several sub test scenarios, we can optimise the corresponding test sequence into a *structured test sequence* consisting of a series of sub test sequences (as shown in Figure 6.4), and each sub test sequence is designed based on its corresponding sub test scenario. Technically, to reduce and control testing complexity with test sequences, it is necessary to perform test sequence optimisation on a complex test sequence that is a long compound sequence consisting of many test artefacts, which may be derived from a complex test scenario captured by a corresponding test model. One effective way of optimising a complex test sequence is to appropriately decompose it into a sequence of logically-related test groups. A *test group* (TG) refers to a small or minimal test sequence composed of closely-related test artefacts for a particular testing objective. All constituent test groups should jointly function in an equivalent manner to the original test sequence to uphold the overall testing requirement and integrity. In the same way, a test group may further be divided into smaller test groups as needed.

For the overall CPS TUC1 test sequence shown in Figure 6.3, we can conduct further test sequence optimisation. We can divide it into a sequence of seven basic test groups to create the structured test sequence (as illustrated in Figure 6.5), where sub test sequence #1 contains the first two basic test groups, sub test sequence #2 contains the middle three basic test groups and sub test sequence #3 contains the last two basic test groups. Each *basic test group* contains at least one specific *verifiable test contract* for a particular testing objective, and may be numbered with its main test contract's number. For example, basic test group 1.2 TG contains test contract 1.2 ITC, and verifies whether the traffic light device is in the expected state of "TL_GREEN" before the test car enters the PAL entry point. Basic test group 3.2 TG contains test contract 3.2 ITC, and verifies whether the traffic light device is in the expected state of "TL_RED" after the test car has entered the PAL entry point. We can also combine two or more basic test groups into a new *joint test group* for a particular joint testing purpose (which is to be further discussed in Section 6.5.3).

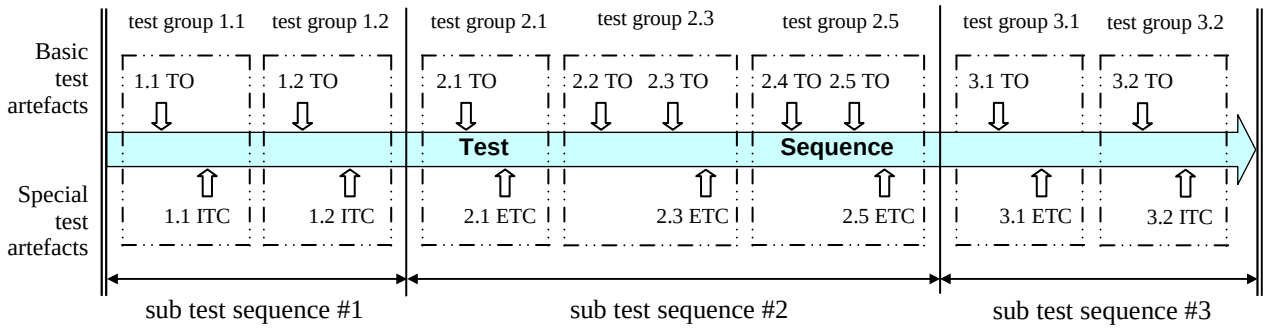


Figure 6.5 Structured Test Sequence = a sequence of test groups (CPS TUC1 Test Scenario)

6.5.2 Test Design for Verifying Component Interactions with Test States

This section discusses how to undertake contract-based component test design to examine component/object interactions by verifying particular test operations, test contracts and associated test states for CIT, including inter-object integration testing and inter-component integration testing. For the CIT purpose, we apply the TbC technique and TbC test contract criteria, and use well-designed test contracts (including ITCs and ETCs) to trace and examine dynamic changes of interacting object states against certain expected *test states*. The test states are used as the testing basis for *test oracle* design for test evaluation (e.g. evaluating whether a component/object retains the expected state when its related operation is performed), and are incorporated into contract-based component test design to examine whether one or more related interacting object operations are performed correctly for the corresponding object interaction. Table 6.3 shows the relationship between test contracts and test operations (with specified signatures) as well as test states, which are used for contract-based component test design for conducting CIT in the CPS TUC1 integration testing scenario (as shown earlier in Figure 5.4).

As an essential requirement for the CIT purpose, test design needs to cover sufficient testing-required component/object operations participating in SCI, which is based on effective test model development that bridges the "test gap" (especially *Test-Gap #1*) in model-based testing (as described earlier in Sections 5.2.4.2 and 5.5.2). For the CPS TUC1 integration testing scenario, Table 6.3 comprises all associated parking control operations of the related CPS control devices (i.e. the traffic light and in-PhotoCell sensor devices) and car movements along the PAL (i.e. making a total of 9 test operations shown in the "Test Operation" column). This ensures that our test design can exercise the necessary component/object operations participating in the CPS TUC1 integration testing scenario.

On the above the basis, contract-based component test design needs to cover adequate test contracts that are applied to all testing-required component/object operations for effective CIT, which is based on effective test model development that bridges the "test gap" (especially *Test-Gap #2*) in model-based testing (as described earlier in Sections 5.2.3, 5.2.4.2, 5.4.3 and 5.5.3). For the CPS TUC1 integration testing scenario, Table 6.3 (in the "Test Contract" column) comprises the necessary test contracts that are applied to all parking control operations for providing parking control services, in order to verify the changes in the related control states (which are the test states for the CPS system, as shown in Table 6.3 "Test State" column).

In the following, by using some selected testing examples in the CPS TUC1 integration testing scenario, we illustrate how a specific ITC/ETC is identified and created for contract-based component test design, and used to conduct contract-based CIT with test states. The testing shows that component test design is actually undertaken based on test sequence design (e.g. designing the structured test sequence with test groups, as described in Section 6.5.1).

Table 6.3 Contract-Based Component Test Design (CPS TUC1 Test Scenario): test sequences, test groups, test operations, test contracts and test states

Test Sequence	Test Group	Test Operation	Test Contract	Test State
enter PAL		enterAccessLane()		
			0.1 ITC: checkState(stoppingBar, "SB_DOWN")	SB_DOWN
Sub Test Sequence **#1** turn Traffic Light to GREEN		1 TS: turnTrafficLightToGreen()		
	1.1 TG	1.1 TO: waitEvent(stoppingBar, "SB_DOWN")	1.1 ITC: checkEvent(stoppingBar, "SB_DOWN")	SB_DOWN
	1.2 TG	1.2 TO: setGreen()	1.2 ITC: checkState(trafficLight, "TL_GREEN")	TL_GREEN
Sub Test Sequence **#2** enter the PAL entry point		2 TS: enterAccessLan()		
	2.1 TG	2.1 TO: waitEvent(trafficLight, "TL_GREEN")	2.1 ETC: checkEvent(trafficLight, "TL_GREEN")	TL_GREEN
	2.3 TG	2.2 TO: goTo(gopace-cross-inPC, int)		
		2.3 TO: occupy()	2.3 ETC: checkState(inPhotoCell, "IN_PC_OCCUPIED")	IN_PC_OCCUPIED
	2.5 TG	2.4 TO: goTo(gopace-crossover-inPC, int)		
		2.5 TO: clear()	2.5 ETC: checkState(inPhotoCell, "IN_PC_CLEARED")	IN_PC_CLEARED
Sub Test Sequence **#3** turn Traffic Light to RED		3 TS: turnTrafficLightToRed()		
	3.1 TG	3.1 TO: waitEvent(inPhotoCell, "IN_PC_CLEARED")	3.1 ETC: checkEvent(inPhotoCell, "IN_PC_CLEARED")	IN_PC_CLEARED
	3.2 TG	3.2 TO: setRed()	3.2 ITC: checkState(trafficLight, "TL_RED")	TL_RED

(1) *ITC Example*

Test design constructs test group 1.2 TG composed of test operation 1.2 TO `setGreen()` and test contract 1.2 ITC `checkState( trafficLight`, "TL_GREEN"), which works as follows:

(a) This test contract checks whether the traffic light is in the correct state of "TL_GREEN" as expected, after test operation 1.2 TO `setGreen`() is performed.

(b) This test contract is applied to operation `setGreen`() in object `trafficLight` and verified in object `deviceController`. As both objects are within the same scope of the device control component, test contract 1.2 is referred to as an ITC.

(c) This ITC examines a typical object interaction within the scope of a single component.

(d) This ITC by design is side-effect free (as indicated in Section 6.3.2 and Section 6.3.3.4). Specifically, this ITC only checks whether the traffic light is in the expected state of "TL_GREEN", and does not affect or change the current state of the traffic light and any other test artefacts (or testing-related data/values).

Note that we refer to this test contract 1.2 as an ITC in terms of the strict general component context (i.e. within the scope of the device control component), not an individual class context that is only a partial component scope. If an individual class scope (i.e. class `TrafficLight`) is regarded relatively as a basic context for effectual contract scope, test contract 1.2 may also be referred to as an ETC, since it examines an object interaction for inter-class integration testing between two classes (i.e. class `TrafficLight` and class `DeviceController`), but these classes are all within the scope of the same single component (i.e. the device control component).

(2) *ETC Example*

Test design can construct test group 2.3 TG composed of test operation 2.3 TO `occupy`() and test contract 2.3 ETC `checkState`(`inPhotoCell`, "IN_PC_OCCUPIED"), which works as follows:

(a) This test contract checks whether the in-PhotoCell device is in the correct state of "IN_PC_OCCUPIED" as expected, after test operation 2.3 TO `occupy`() is performed.

(b) This test contract is applied to operation `occupy`() in object `inPhotoCell` in the device control component, but is verified in object `testCarController` in the car control component. So test contract 2.3 is referred to as an ETC.

(c) This ETC examines a typical component interaction for inter-component integration testing between the two CPS collaboration components.

(d) This ETC by design is side-effect free (as indicated in Section 6.3.2 and Section 6.3.3.4). Specifically, this ETC only checks whether the in-PhotoCell device is in the expected state of "IN_PC_OCCUPIED", and does not affect or change the current state of the in-PhotoCell device and any other test artefacts (or testing-related data/values).

6.5.3 Test Design for Verifying Component Interactions with Test Events

In this section, we explore another important aspect of CIT: we design contract-based tests with test events to verify particular object interactions by checking certain communication messages (or *event communications*) that realise the object interactions between collaborating objects. We illustrate this type of contract-based component test design by retesting the Observer pattern-based component `EventCommunication` that is reused in the CPS TUC1 integration testing context, after this base component has been tested in its unit testing context. To carry out this CIT task, we conduct test design with special test contracts to examine and verify certain event communications by checking particular *test events*, in order to ensure that the specific event communication is correctly performed in the SCI context. For example, test design is required to be able to verify whether the registered event listener receives the correct event notification from the correct event notifier as described in the Observer pattern [63]. For the CPS TUC1 integration testing scenario, this testing is especially important when system control shifts from the device control component to the car control component at the control switchover point, and vice versa.

In the following, we illustrate how test design constructs and applies a special *joint test group* of related test contracts and test operations (as shown in Figure 6.6) to examine a particular *test event* to ensure that system control is shifted correctly between (1) the device control component and (2) the car control component at the control switchover point. This special joint test group actually combines two basic test groups: test group 1.2 TG in sub test sequences #1 and test group 2.1 TG in sub test sequences #2 (note that these sub test sequences and test groups were designed in Section 6.5.1, as shown in Figure 6.5). Also this special joint test group crosses over from sub test sequences #1 to sub test sequence #2 to cover the control switchover point in the CPS TUC1 integration testing scenario.

(1) In sub test sequence #1 of the CPS TUC1 integration testing scenario, system control commences with the device control component prior to the control switchover point. The component controls parking operations approaching the control switchover point:

(a) Test operation 1.2 TO `setGreen`() (from the basic test group 1.2 TG) runs on object `trafficLight` to set the traffic light to the new state of "TL_GREEN" for the next car's access to the PAL.

(b) The execution of this test operation causes the object's state change, which results in a new event being generated. Then by conducting an event communication with the Observer pattern-based component `EventCommunication`, the event notifier object `trafficLight` needs to notify the new event to all of its waiting event listener objects `testCarController` and `deviceController` for the control switchover.

(c) Like test design with test states (as described in Section 6.5.2), test contract 1.2 ITC `checkState( trafficLight`, “TL_GREEN”) (from the basic test group 1.2 TG) is constructed as an ITC to check whether the traffic light device is now in the correct state of “TL_GREEN” as expected in the scope of the device control component, before the system control is switched over.

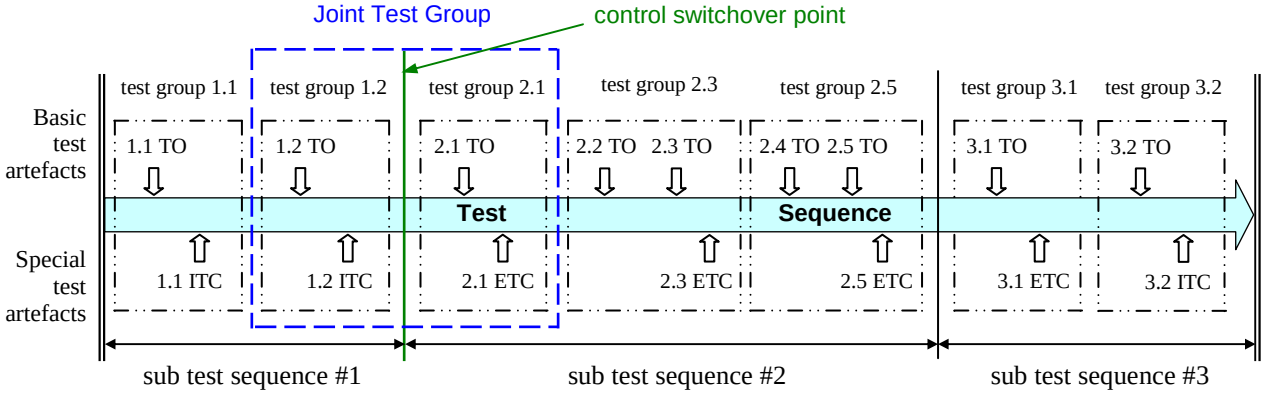


Figure 6.6 Contract-Based Component Test Design: joint test group for CIT (CPS TUC1 Test Scenario)

(2) Then, system control shifts to the car control component (accordingly, the testing shifts from sub test sequence #1 to sub test sequence #2):

(a) The waiting car (as the event listener object `testCarController`) waits for an incoming event notification as a parking instruction to assess the PAL. This is conducted by test operation 2.1 TO `waitEvent( trafficLight`, “TL_GREEN”) (from the basic test group 2.1 TG) running on object `testCarController`.

(b) When the event communication is fulfilled with the base component `EventCommunication`, the car needs to take some action according to the received event notification. However, before the car enters the PAL, it is necessary to recheck whether the event reception is correct on the event listener object `testCarController`. Test contract 2.1 ETC `checkEvent( trafficLight`, “TL_GREEN”) (from the basic test group 2.1 TG) is constructed as an ETC to check whether the waiting car (i.e. the event listener object `testCarController` in the car control component) receives the correct event notification (i.e. the traffic light is in the correct state of “TL_GREEN”; the car is allowed to enter the PAL) from the correct event notifier object `trafficLight` in the device control component.

(c) When the completion of this event communication between the two CPS components is checked to be correct, the system control switchover is correct. Then, the car starts entering and accessing the PAL with a sequence of related parking operations controlled by the car control component.

6.6 Related Work and Discussion

This section reviews and discusses research work particularly related to contract-based testing in line with the DbC principle. This serves as an extended literature review specific to the TbC technique, which is based on the foundation literature review as described earlier in Chapter 2 and Chapter 3.

Beugnard et al. [22] define a general model of software contracts at four levels: basic or syntactic contracts, behavioural contracts, synchronisation contracts and quality-of-service contracts. Because behavioural contracts are more pertinent to the DbC principle in component design and testing practice, our TbC technique promotes well-designed test contracts particularly as behavioural contracts for UML-based CIT. Briand et al. [31] investigate analysis contracts to improve the testability at the level of object-oriented code. Their contract definition rules mainly apply to the class unit context, and analysis contracts are expressed in OCL [160]. They also use contract-related instrumentation tools to instrument contracts for their testing example of the ATM system, and evaluate relevant testability features, benefits and limitations. Edwards et al. [56] present a contract wrapper approach to enhance component testing capabilities for component functional testing, without access to the low-level details inside component code. This approach is more flexible for improving design-based component testability, and offers good testing benefits for both component developers and users. However, developing companion test wrappers for all components under test may attract high workloads and costs in testing. Nebut et al. [102] present a use case driven approach to system testing. They build on UML use cases enhanced with contracts based on use case pre and post conditions. System test cases are generated in two steps: use case orderings are deduced from use case contracts, and then use case scenarios are substituted for each use case to produce system test cases.

By comparison, our research with the TbC technique has its own particular characteristics different from other related work, which contributes to the following important aspects:

(1) The TbC technique develops a set of important contract-based test concepts (e.g. test contract, Contract for Testability, effectual contract scope, internal/external test contract), and useful TbC test contract criteria for effective testability improvement at the modeling level (see Sections 6.2 and 6.3).

(2) The TbC technique bridges the “test gaps” and improves model-based component testability for test model construction, and support UML model-based approaches to SCT (see Section 6.4; and Sections 5.2.3, 5.2.4.2, 5.3, 5.4.3 and 5.5.3).

(3) The developed TbC working process guides contract-based testing activities (see Section 6.2), and we have illustrated how to put them into practice for contract-based test design with a case study (see Sections 6.5).

(4) The TbC technique is a direct extension of the DbC concept (which was developed originally for object-oriented design) to the new domain for SCT, and becomes a useful self-contained contract-based approach to SCT (see Sections 6.1 and 6.2).

6.7 Summary

This research has extended the DbC concept to the SCT domain, and developed the TbC technique as a new contract-based SCT technique with a primary aim to bridge the "test gaps" between ordinary UML models (non-testable) and target test models (testable) and improve model-based component testability for effective UML-based SCT. In this chapter, we introduced the new test contract concept as the key testing-support mechanism, and the new concept of Contract for Testability as the principal goal of the TbC technique. We described the test contract concept based on basic component contracts, classified test contracts into internal and external test contracts for effective contract-based testing based on the new concept of effectual contract scope, and developed a set of useful TbC test contract criteria to realise testability improvement for achieving the CfT goals. Then, following the developed TbC working process, we showed how to apply the TbC technique to test contract design for test model construction and contract-based component test design by using the illustrative testing examples selected from the CPS case study. The testing examples have demonstrated that the TbC technique is capable of bridging the identified "test gaps" (especially *Test-Gap #2*), improving model-based component testability and supporting effective component test design. These are some of the major contributions of the TbC technique.

Therefore, this chapter has shown that component test development with the MBSCT methodology is not only model-based, process-based and scenario-based, but also contract-based (note that the relevant MBSCT methodological features will be further justified in Sections 8.2 and 8.5). At the same time, this chapter has employed the TbC technique to demonstrate and validate the MBSCT testing applicability and capabilities particularly for component test design, adequate test artefact coverage, and component testability improvement (which are the core MBSCT testing capabilities #2, #4 and #5 as described earlier in Section 4.6). A more comprehensive validation and evaluation of the MBSCT methodology will be presented in Chapter 9.

This chapter has mainly covered the TbC foundation phase (including Steps TbC1, TbC2 and TbC3) in the stepwise TbC working process (as shown in Figure 6.1). The TbC advanced phase (including Steps TbC4 and TbC5) will be discussed in the subsequent chapters of this thesis. Chapter 7 will describe component fault detection and diagnosis with the TbC technique (i.e. Step TbC4). Contract-based test generation (i.e. Step TbC5) will be discussed in Chapter 8.

Chapter 7
Component Fault Detection, Diagnosis and Localisation

7.1 Introduction

Component test design aims to detect and diagnose component faults for the goal of enhancing and assessing component reliability and quality (see Section 7.2). At the same time, component fault detection and diagnosis (FDD) is a useful means to improve and evaluate the effectiveness of component test design with a testing approach. We undertake component fault detection and diagnosis as an integral part of component test design in the Phase #2 of the MBSCT framework. With the MBSCT methodology, FDD is model-based, which means that FDD is undertaken with test models and model-based component tests. FDD is also scenario-based, which means that test scenarios are used as the basis to detect and diagnose target component faults in the related component functional scenario. Moreover, FDD is contract-based, which means that the TbC technique plays a key role in the process of component fault detection, diagnosis and localisation, and this process is undertaken jointly with our contract-based component test design (CBCTD) approach (as described earlier in Section 6.5).

Chapter 6 presented the foundation principles of the TbC technique, and described the foundation phase (including Steps TbC1 to TbC3) in the stepwise TbC working process (as shown earlier in Figure 6.1 in Section 6.2). This chapter moves on to the advanced phase in the stepwise TbC working process, and focuses on the testing progression from Step TbC3 to Step TbC4 with the TbC technique. In particular, we focus component test design on component fault detection and diagnosis with the TbC technique. For this purpose, we develop a new *contract-based fault detection and diagnosis* (CBFDD) method [173] [175] [176], which further extends the TbC technique to support effective SCT and establishes a key technical foundation for component test evaluation (see Chapter 9).

This chapter presents component fault detection, diagnosis and localisation with the TbC technique to achieve effective component test design. First, Section 7.2 describes some important fault-related terms and their relationships, and presents an extended fault causality chain to guide SCT activities in FDD and effective component test design. Section 7.3 introduces a new important notion of *Contract for Diagnosability* (CfD) to be a key objective of our CBFDD method, and this notion particularly satisfies the higher-level goals of the *Contract for Testability* concept with the TbC technique (as described earlier in Section 6.3). In Section 7.4, we develop a practical CBFDD process for fault detection, diagnosis and localisation, which is a ma-

jor technical component of the CBFDD method. In Sections 7.5.1 to 7.5.4, we analyse and explore certain critical inter-relationships between test contracts and fault diagnosis properties in terms of *effectual contract scope*, *fault propagation scope*, and *fault diagnosis scope*, which are new testing notions we introduce to support the CBFDD method. In Section 7.5.5, we develop the stepwise upper/lower-boundary scope reduction strategies and processes, and provide the useful CBFDD guidelines for effective fault diagnosis and localisation. The CBFDD guidelines are another major technical component of the CBFDD method. Then in Section 7.6, we apply the CBFDD method, and employ the CPS case study to illustrate how to put the CBFDD method into practice to undertake component fault detection, diagnosis and localisation. We develop the two useful diagnostic solutions with the CBFDD method in the two major possible testing contexts (see Sections 7.6.2.2 and 7.6.2.3). Section 7.7 discusses some important open issues and defines a set of new useful notions related to the TbC technique (especially the CBFDD method). Section 7.8 summarises this chapter.

7.2 Fault Causality Chain: Fault → Error → Failure

A primary reason why software testing is required is that any activity during the software development process may introduce or produce certain software defects or imperfections in the developed software. This contributes to a principal objective of software testing that aims to detect and uncover such software defects and imperfections in the software/system under test (SUT) as much as possible, in order to improve and evaluate software reliability and quality. For use in the testing process, it is important to clarify several important terms and their relationships: fault, error, and failure [77].

Failure refers to a manifested incapacity to function or perform satisfactorily. It means some undesired behaviour observed during the execution of the SUT, which unsuccessfully meets the expected objective, e.g. there is some incorrect output or inability of fulfilling the expected functional requirement.

Fault refers to a defect or imperfection in the SUT. A fault may be a malfunction, imperfect data or operational definition, incorrect execution or operating step/process, etc. It is created by certain incorrect activities during the software development and/or operation process. A fault may also remain inactive and undetected, and thus may have no impact on the SUT or other related interacting software or systems even for a long time.

Once a fault is encountered and activated by software execution, it can negatively impact on the SUT to produce a certain undesirable state or incorrect operational manifestation, which is called an *error* (or a corrupted state). As the error develops and propagates to an incorrect output, it may result in a subsequent failure of software execution. An error is an intermediate corrupted or incorrect state between the original fault and the resulting failure, and may occur

internally in the SUT.

Accordingly, we can develop the following "*fault causality chain*" to illustrate a causality relationship among these three defect counterparts as shown in Figure 7.1, which is a further extension from the work by Avizienis et al. [6] [7] [8]. This fault causality chain describes a typical cause-effect relation: an activated fault produces an error, which, by propagation, subsequently causes a failure. Faults are the original sources of failures, and failures are the negative outcomes resulting from faults. Note that the causality relationship can also iterate recursively with the dashed line between failure and fault (as shown in Figure 7.1). In particular, a fault may be subsequently caused by the failures of other related interacting software or systems [18].

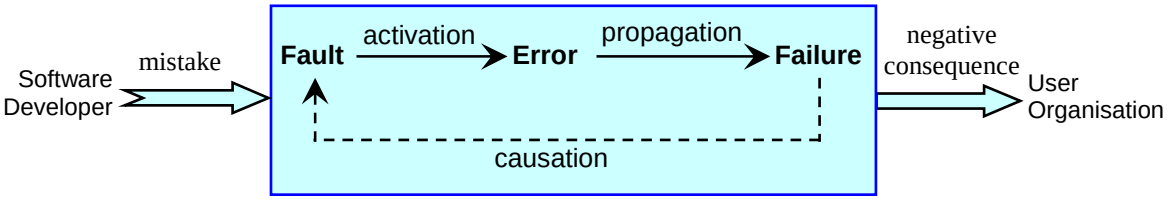


Figure 7.1 An Extended Fault Causality Chain

From this extended fault causality chain, we can see that both software developer and user organisation are the external stakeholders. Software activities by software developers may make possible mistakes, which is a source input to initialising the fault causality chain. The user organisation receives the possible negative consequences, which is the ultimate output resulting from the fault causality chain.

Furthermore, we can analyse and explore the following important implications from the extended fault causality chain:

(1) It is the software activity by the software developer that incorrectly creates software faults during software development;

(2) It is the execution of the software fault that causes actual software failures;

(3) It is the software failure resulting from software faults that damages software system operations and organisation business operations.

(4) A single fault can cause multiple failures, although some faults may never turn into failures. On the other hand, the same failure may be caused by different faults with different software execution patterns at different times.

(5) The primary concern with faults is that faults can develop into failures and produce negative impacts, whenever a fault is active and the software segment that contains the fault is executed.

The above extended fault causality chain and related important implications are particularly useful to guide SCT activities in FDD and effective component test design. In testing practice, the tester needs to design effective component tests that are able to activate a certain component fault to cause some observable manifestation of failure. From the observed component failure, the tester needs to track down and analyse possible component errors, and identify and reveal the original component fault, in order to correct the fault. Therefore, a central task of SCT is to design and generate component tests that can detect and diagnose component faults effectively and efficiently. This is one of the principle goals of our research on SCT.

7.3 Contract for Diagnosability

The TbC technique employs the test contract mechanism and test contract criteria to achieve the CfT goals in two important aspects: not only constructing adequate test contracts for *testability specification and improvement*, but also conducting effective component test design for *testability verification and evaluation* (as described earlier in Section 6.3). The second CfT aspect particularly supports the goal of *Contract for Diagnosability* (CfD): the TbC technique aims to undertake CBCTD (as described earlier in Section 6.5) that can effectively detect and diagnose component faults, and evaluate and demonstrate the required level of component correctness and quality. Therefore, our CBCTD aims to not only design and generate component tests with fault detection capability, but also diagnose and locate the detected faults for correction and removal, which is a primary goal of our CBFDD method.

According to the test contract principle (as described earlier in Section 6.3.1), violating the required contracts of the mutual responsibilities bound by both component partners (component service supplier/contractor and client) indicates the presence of possible component faults, which often results from incorrect component design and specification (e.g. incorrect UML-based model specification). On the other hand, based on the principle of the fault causality chain (fault-error-failure) (as described in Section 7.2), it is a *component fault* that produces an intermediate *component error*, which, by propagation, subsequently causes a *component failure*, which could finally result in an incorrect and unsatisfactory component integration in a component-based system.

From the above analysis based on the test contract principle and the fault causality chain, we can see that it is not adequate to conduct a simple component fault detection that only reveals and shows some faults present during testing. If the detected fault is not located and corrected, the same fault or its variants will still exist and continuously cause the same or similar software failures during software execution or testing. An effective SCT technique should have effective fault-diagnosis capabilities that are able to diagnose and locate the detected fault for

correction and removal. Technically, *fault diagnosis* denotes the testing process that analyses fault cases and causes, and identifies and locates the detected fault in the associated faulty part of the component under test (CUT), when a failure is observed due to the detected fault during testing. In the TbC context, the *Contract for Diagnosability* feature denotes the testing capability for identifying and locating the detected fault with well-designed test contracts for the goal of effective fault diagnosis and localisation. A key measure of a good CBCTD is that it should be able to support and realise the CfD goal effectively. Our CBFDD method particularly focuses on fault diagnosis that bridges fault detection and fault localisation. In other words, our CBFDD method covers not only the basic capability for fault detection and diagnosis, but also the advanced capability for fault diagnosis and localisation.

Based on test models constructed with the TbC technique, our CBCTD can combine relevant adequately-designed test contracts and test operations together into particular *test sequences* or *test groups* (as described earlier in Section 6.5) to detect and diagnose possible component faults in the CIT context. In particular, if the assertion of a test contract in the current CBCTD returns *false*, a component fault has probably occurred and so has been detected during testing, and the fault is most likely related to the associated operation under test that is involved in the testing scope of the current CBCTD. Furthering this key strategy of CBCTD, we focus our CBFDD method on the following important technical aspects to realise the CfD goal:

(a) Developing a systematic process to effectively guide component fault detection, diagnosis and localisation, which we call the CBFDD process that becomes a major technical component of the CBFDD method (see Section 7.4);

(b) Exploring and analysing certain intrinsic relationships between test contracts and fault diagnosis properties to improve test design quality (see Sections 7.5.1 to 7.5.4); and then

(c) Developing the related scope reduction strategies and processes, and providing useful technical guidelines for effective fault diagnosis and localisation, which we call the CBFDD guidelines that become another major technical component of the CBFDD method (see Section 7.5.5).

7.4 Contract-Based Fault Detection and Diagnosis Process

With the TbC technique, we develop a practical CBFDD process that involves five main steps in conjunction with fault case analysis, which is illustrated in Figure 7.2. A major purpose of the CBFDD process aims to systematically guide fault detection, diagnosis and localisation effectively with CBCTD (as described earlier in Section 6.5). This process establishes the primary foundation of our CBFDD method to realise the CfD goal.

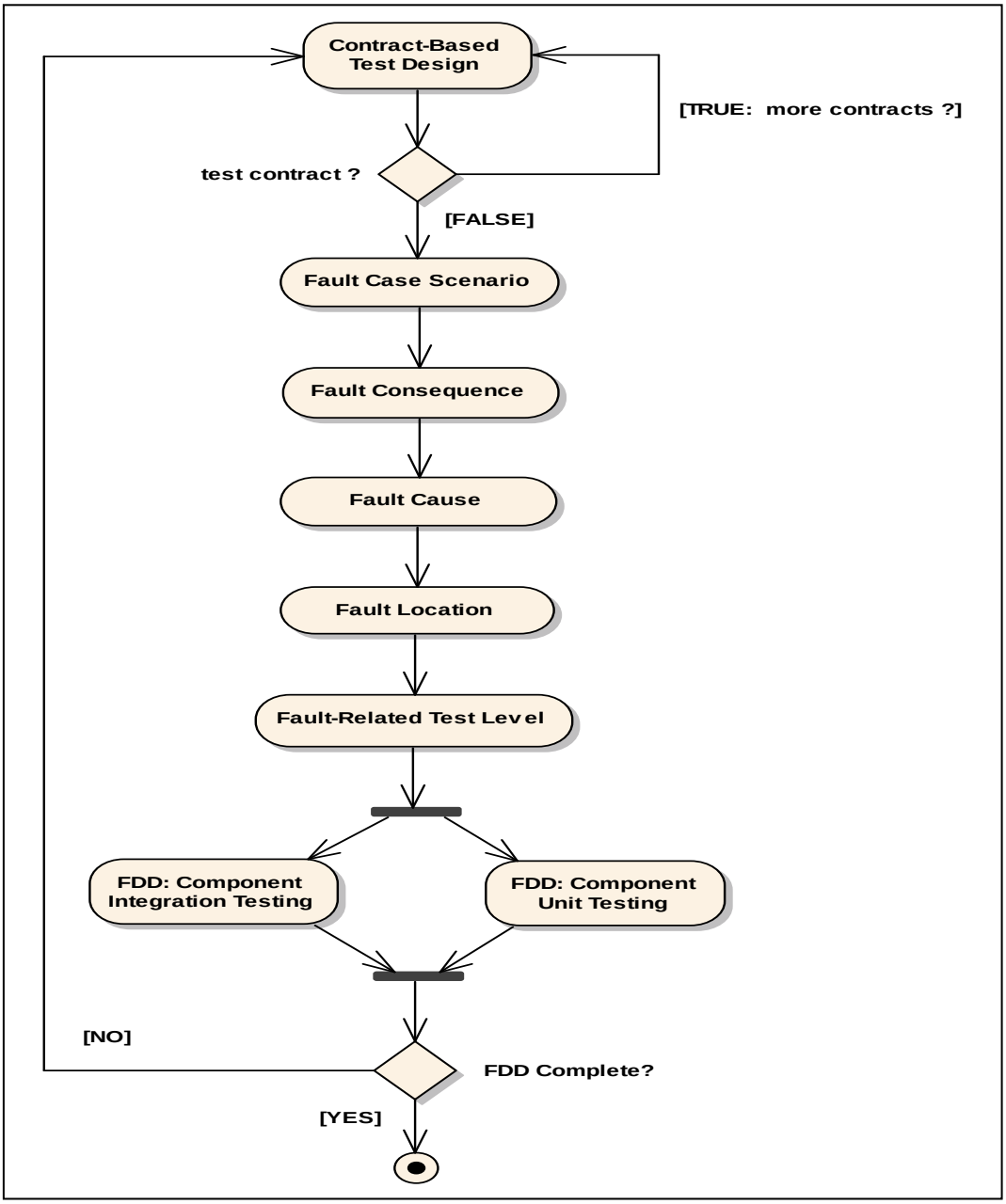


Figure 7.2 Contract-Based Fault Detection and Diagnosis Process

The following describes the main technical aspects of the CBFDD process for the CIT purpose:

(1) *Fault case scenario*

When the test contract returns *false* (i.e. a contract violation occurs), a component fault has been detected during testing with the current CBCTD. We need to analyse the observed failure scenario and diagnose what has happened to the related failure output in order to diagnose and locate the detected fault.

(2) *Fault consequence*

We need to analyse what consequence might have resulted from the contract-violated failure output. The relevant consequence includes all possible direct and indirect negative impacts on the CUT in the CIT context.

For example, suppose the current component operation under test fails the completion of an expected component function, then this negative outcome may further cause some subsequent operations not to be executed as needed in the expected sequence of software operations, or potentially the entire CBS execution could be halted unexpectedly at this failure point in the CIT context.

(3) *Fault causes and analysis*

Based on the analysis of the fault case scenario and consequence, we need to further determine possible causes according to the principle of the fault causality chain (fault-error-failure) (as described in Section 7.2). In particular, we analyse and uncover what possible errors are made during the fault propagation process towards the failure point.

Typically, possible fault causes may include:

(i) Fault cause #1: the incorrect invocation/usage of a specific operation that is being exercised and examined by the current CBCTD; or

(ii) Fault cause #2: the incorrect definition/implementation of this operation in its home class unit.

(4) *Fault location*

When the possible fault cause is determined, we are then able to identify the possible software location of the fault under diagnosis:

(a) Fault location for incorrect invocation/usage: For the above fault cause #1, the fault under diagnosis is most likely located in the caller component class (e.g. it may be an integration control class for component integration purpose, and serves as the current integration context), which incorrectly invokes and uses that specific operation under test.

(b) Fault location for incorrect definition: For the above fault cause #2, the related fault is most likely located in its home class unit, where the operation is incorrectly defined and/or implemented.

(5) *Fault-related test level*

The two possible fault locations as described above may be in the same home component/class or possibly across multiple different components/classes, depending on the nature of the fault under diagnosis. The two different fault locations by their nature indicate that the possible fault occurrence is pertinent to the two different levels of SCT:

(a) If the fault is located in the integration class, then this fault occurrence is clearly related to *inter-class* or *inter-component integration testing*, because the fault occurs when conducting component integration via operation invocations for object interactions and collaborations in the SCI context.

(b) If the fault is located in the home class unit, then this fault occurrence is clearly related to *class unit testing*, because the fault occurs when defining the class unit (e.g. defining class operations and attributes). If so, this indicates that the previous component unit testing has not been sufficiently adequate before the testing proceeds to CIT.

(c) It is observed that, because the failure output is caused by this detected and located fault, the related test contract eventually returns *false* in the above fault case scenario (as described in (1) above).

By effectively applying the CBFDD process, the TbC technique can aid our CIT approach to achieve two testing benefits: we can examine and detect possible component faults that are related to not only certain integration contexts as the central focus of CIT, but also to certain component class units as a secondary focus of CIT. Any component faults uncovered in class units require undertaking more component unit testing for the purpose of effective CIT performance. All detected/located component faults need to be corrected and removed, and necessary regression testing needs to repeat the related integration/unit testing activities after the software modification for fault correction and removal.

The CBFDD process provides an overall FDD process with CBCTD. A key focus of the CBFDD process is on how to design and apply appropriate test contracts for effective fault diagnosis and localisation, which is further discussed in Section 7.5 below. The CBFDD process is an iterative and incremental testing process, and has the following characteristics:

(1) The CBFDD process starts with CBCTD, and when some test contract with the current CBCTD detects a component fault occurrence through the contract-violated failure output, the steps of the CBFDD process are applied to diagnose and locate this detected component fault with the current CBCTD.

(2) The CBFDD process can be used to detect and diagnose new potential component faults when additional test contracts are *incrementally* designed and added to the current CBCTD to meet a new testing objective. This particularly accommodates and supports the perspectives and needs of the component tester, who must identify and uncover as many potential component faults as possible.

(3) The current CBCTD may need additional test contracts that are constructed *incrementally*

in the CBFDD process, in order to detect and diagnose a specific target component fault.

(4) The CBFDD process should be re-conducted *iteratively* with the required regression testing, after a fault is identified and fixed with the CBFDD process undertaken previously, in order to ensure that the previously detected fault is ultimately corrected and removed.

(5) The CBFDD process starts with CBCTD, and works *iteratively* and *incrementally*, until all FDD tasks have been completed to fulfil the target testing requirements (e.g. meeting the required level of component correctness and quality).

7.5 Fault Detection, Diagnosis and Localisation

The effectiveness and efficiency of CBFDD mainly depends on the quality of CBCTD, which is determined typically by the quality of test contracts developed with the TbC technique for CBCTD. Clearly, a good CBCTD is required to be able to detect and diagnose certain target component faults with adequately-designed test contracts. To improve the CBCTD quality for the CfD goal, we need to further explore certain critical relationships between test contracts and fault diagnosis properties. With the TbC technique, we focus on three important notions and their intrinsic relationships for supporting our CBFDD method: fault propagation scope, fault diagnosis scope, and effectual contract scope (as shown in Figure 7.3). A key purpose of the CBFDD method is to examine how to discover and use these key notions and their relationships to guide test contract design effectively to facilitate fault detection, diagnosis and localisation, which is a major focus of the CBFDD process (as described in Section 7.4). The following subsections discuss relevant concepts, technical aspects and guidelines for fault diagnosis and localisation with test contracts.

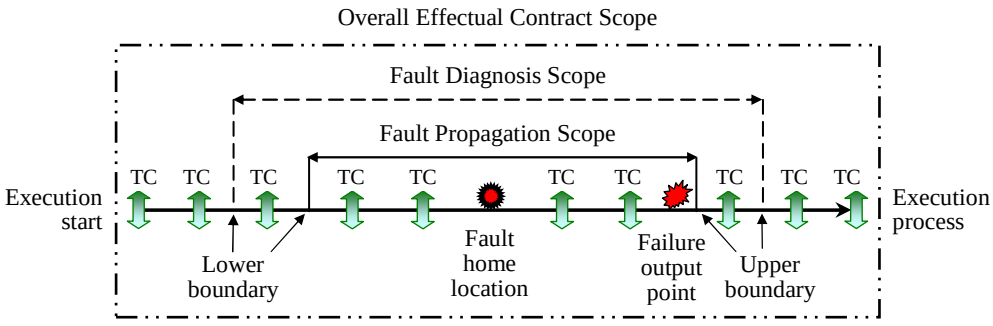


Figure 7.3 CBFDD: Test Contracts and Fault Diagnosis Properties

7.5.1 Fault Propagation Scope

During testing, a possible component fault occurrence may not be noticed until the main or final system outputs produce a failure. Because the fault propagation process (fault-error-failure) (as described in Section 7.2) usually spans a space from its start to its end, there exists a certain software artefact range from the fault's original home location to the actual failure output point, which becomes a *fault propagation scope*.

The notion of fault propagation scope has an impact on fault diagnosis and localisation, which needs to undertake the following important CBFDD activities for the purpose of effective FDD (as illustrated in Figure 7.3):

(a) *Delimit* the possible (maximum) boundary of the relevant fault propagation scope;
(b) *Constrain* the fault propagation scope within the delimited boundary;
(c) *Reduce* the range of the relevant fault propagation scope in order to facilitate fault diagnosis and localisation.

At the initial stage, the maximum scope of fault propagation would range from the execution start point to the final failure output point. Depending on the actual software execution scenarios, the actual fault propagation scope may vary even for the same fault, and could extend across different classes, different components or different SCI scenarios. It is observed that such uncertainty in the fault propagation scope is one of the key reasons why it is very difficult to exactly identify, diagnose and locate a specific fault in testing practice. Moreover, because the exact failure output point may actually be unknown, the possible final failure output point could be just at the last execution point of the SUT in the worst-case situation. This means that the *maximum fault propagation scope* may range from the first execution point to the last execution point.

7.5.2 Fault Diagnosis Scope

A test case can be regarded as successful if it detects an as-yet undiscovered error/failure of a system or component [100]. This accordingly indicates that there exists a possible new component fault that has not been detected before, or there still exists a previous component fault that has not been corrected and removed yet. However, whichever fault type occurs, it could lead to a new failure. An ordinary test case usually has a testing range where it exercises and examines the possible fault, although it may not be able to determine the exact location of the fault. Such a testing range encloses a software artefact context (e.g. a component context or modeling context) where the fault most likely exists, which becomes a *fault diagnosis scope*.

The notion of fault diagnosis scope also impacts on fault diagnosis and localisation, which requires similar CBFDD activities for the purpose of effective FDD (as illustrated in Figure 7.3) as follows:

(a) A basic requirement for fault diagnosis and localisation is that the fault diagnosis scope must *cover* the relevant fault propagation scope to finally identify and locate a specific related fault.

(b) A further requirement for fault diagnosis and localisation is that it should be able to *delimit* and *constrain* the possible boundary of the relevant fault diagnosis scope, and then *reduce* the relevant fault diagnosis scope, and thus facilitate diagnosing and locating a specific component fault in the FDD process.

The notion of fault diagnosis scope is a very useful mechanism to control and deal with the relevant fault propagation scope. There are several situations of fault diagnosis scope that can be applied to actual fault diagnosis and localisation:

(1) At the initial stage, the *initial scope* of fault detection and diagnosis ranges from the execution start point to the final failure output point. To deal with the maximum fault propagation scope in the worst-case situation (as described in Section 7.5.1), the *maximum fault diagnosis scope* needs to cover the range from the first execution point to the last execution point.

(2) The fault diagnosis scope may be a particular *test scenario* that delimits and constrains the possible range for fault propagation scope. Test scenarios are a useful means to isolate a possible fault-related testing range from the other parts of the SUT outside the current testing context, so that FDD can focus on a particular test scenario that covers the fault-related testing range.

(3) The fault diagnosis scope may be a particular *component* under test, in which the fault under diagnosis originally occurs and propagates. This is a fault diagnosis scope at the component level.

(4) The fault diagnosis scope may be a specific *component unit* (e.g. a class of the CUT), which is the home location of the fault under diagnosis. This is a fault diagnosis scope at the component unit level.

(5) The fault diagnosis scope may be related to a specific component/class *operation* whose definition and/implementation contains the fault under diagnosis. This is a fault diagnosis

scope at the component operation level, which would be the minimum scope of fault detection and diagnosis for component functional testing.

7.5.3 TbC Test Contract Criteria and Fault Diagnosis

The TbC technique provides useful test contract mechanisms and test contract criteria to enable CBCTD to support and facilitate CBFDD. Following the TbC test contract criteria for the CfD goal, CBCTD can design and construct appropriate test contracts to trace execution information of possible component operations and elements, observe and control certain possible failure information and testing points (as described earlier in Section 6.3.5) , in order to detect and diagnose possible component faults. If necessary, CBCTD can also use test contracts to raise appropriate warnings or exceptions at certain key testing points to prevent and stop fault propagation development. Using this typical TbC test contract criteria based FDD approach, component test design developed with adequate test contracts is able to delimit, constrain and reduce the relevant fault propagation scope. Accordingly, the relevant fault diagnosis scope is also delimited, constrained and reduced.

Technically, the above component test design approach to FDD employs the TbC test contract criteria for adequate test contract coverage, which promotes and supports a high level of coverage of adequate test contracts that are applied to all possible component operations and elements under test (as described earlier in Section 6.3.4). This strategy seems straightforward and works to detect and diagnose possible component faults. However, this approach also has some deficiencies. As discussed earlier in Section 6.3.4.8, this test design approach would probably lead to higher testing overheads and an unattainable level of test contract coverage, and thus become impractical for higher-complexity software components and systems under test. In addition, this approach has low testing performance and efficiency for uncovering a specific component fault currently associated with a particular testing objective. This is because not all test contracts or related test artefacts applied based on the TbC test contract criteria are equally effective in reducing the relevant fault diagnosis scope and locating a specific target component fault. Only some of the closely related test contracts contribute to actual diagnosis and localisation of the specific target component fault. Therefore, a balanced trade-off between test contracts and FDD requires that the number of test contracts needed for CBCTD should be as minimal and as adequate as possible to detect and diagnose a specific target component fault effectively and efficiently. This is one of the key features of good CfD practice.

7.5.4 Effectual Contract Scope and Fault Diagnosis

A major goal of our CBFDD method aims to provide an alternative useful approach to overcome some deficiencies of the above TbC test contract criteria based FDD approach (as described in Section 7.5.3 above), and to achieve low-overhead test contract coverage/usage and acceptable testing effectiveness and efficiency. For this purpose, let us further explore the interrelationship between test contracts and fault diagnosis. Based on the important concept of *effectual contract scope* defined in the TbC technique (as described earlier in Section 6.3.3), we can further refine and optimise component test design for effective FDD by developing appropriate types of ITCs and ETCs. In principle, the *overall effectual contract scope* of all types of test contracts (ITCs and ETCs) in CBCTD must *cover* the relevant fault diagnosis scope, which also must *cover* the relevant fault propagation scope (as illustrated in Figure 7.3). Furthermore, our CBFDD method is to employ the notion of effectual contract scope to *control* both fault diagnosis properties (i.e. the relevant fault diagnosis scope and fault propagation scope), in conjunction with well-developed ITCs and ETCs.

In our CBFDD method, for a particular testing objective, we may only need to diagnose and locate a specific target component fault with which we are currently concerned. For example, in certain situations, when a specific component fault is the primary cause of the occurrence of other associated side-effect errors or failures, the correction and removal of this specific component fault can lead to the correction and removal of these side-effect errors or failures that are closely associated with this specific component fault before it is fixed. Accordingly, it is important to use a small number of well-designed test contracts (ITCs and ETCs) that are effective and efficient in diagnosing and locating the specific target component fault to support the CfD goal.

7.5.5 Guidelines for Fault Diagnosis and Localisation

Based on the above analysis of test contracts and fault diagnosis properties, we can see that the TbC technique enhances the basic SCT towards fault detection and diagnosis, and further to localisation for fault correction and removal. In this section, to put our CBFDD method into practice, we develop and provide the following useful technical guidelines for effective fault diagnosis and localisation to realise the CfD goal. The CBFDD guidelines refine and detail the fault diagnosis and localisation activities in the CBFDD process (as described in Section 7.4). In particular, the CBFDD guidelines provide the series of technical steps for fault diagnosis and localisation, based on the three important testing notions of fault propagation scope (as described in Section 7.5.1), fault diagnosis scope (as described in Section 7.5.2), and effectual contract scope (as described in Section 7.5.4 and Section 6.3.3). An additional key part of the CBFDD guidelines is to apply the two *stepwise upper/lower-boundary scope reduction strategies* and the as-

sociated *stepwise upper/lower-boundary scope reduction processes*, which are developed to support the CBFDD method. One major objective aims to effectively and efficiently detect and diagnose a specific target fault for a particular testing objective, by applying a smaller number of well-designed test contracts (ITCs and ETCs), positioning them at certain *selected testing points* and covering certain *selected component artefacts* under test, along with the stepwise scope reduction process.

The CBFDD guidelines for fault diagnosis and localisation are outlined in the six main steps shown in Table 7.1, which are further described as follows:

Table 7.1 The CBFDD Guidelines: an Outline

Step #	Step Description
Step #1	Determining test levels: integration testing or unit testing
Step #2	Determining the fault propagation direction for scope reduction
Step #3	Stepwise scope reduction process for reducing the fault propagation scope and the fault diagnosis scope
Step #3.1	The upper-boundary scope reduction process
Step #3.2	The lower-boundary scope reduction process
Step #4	Reducing the fault diagnosis scope to class/operation scope
Step #5	Locating the target fault that has been detected during testing
Step #6	Correcting and removing the detected fault

(1) Step #1: *Determining test levels: integration testing or unit testing*

First, we need to design appropriate ETCs to discover whether the overall effectual contract scope crosses over certain integration classes/component boundaries. If so, these ETCs needed for CBCTD are essentially used to do CIT; otherwise, they are used to do unit testing. The overall effectual contract scope of these ETCs is the basis for the determination of the initial overall fault diagnosis scope. As described in Sections 7.5.1 and 7.5.2, the initial (maximum) fault diagnosis scope would range from the execution starting point to the final failure output point, which properly covers the initial (maximum) fault propagation scope.

(2) Step #2: *Determining the fault propagation direction for scope reduction*

Around the final failure output point, inserting appropriate test contracts at certain testing points in the relevant (sequential) execution path can ascertain the direction of fault propagation development. Our approach is as follows: we apply appropriate test contracts to raise related warnings or exceptions at certain crucial testing points, which is a useful way to *stop* the development of fault propagation. If a test contract can stop fault propagation development in the relevant execution path before the final failure output point, such an observed outcome indicates the direction of fault propagation. When the fault propagation direction is determined, the rele-

vant test contracts must be inserted at certain testing points that are opposite to the direction of fault propagation development, between the execution start point and the final failure output point, in order to reduce the relevant fault propagation scope and fault diagnosis scope for fault diagnosis and localisation.

(3) Step #3: *Stepwise scope reduction process for reducing the fault propagation scope and the fault diagnosis scope*

Because the possible fault location should exist within the relevant fault diagnosis scope that covers the relevant fault propagation scope (as described in Section 7.5.2), our major approach to scope reduction for the purpose of fault diagnosis and localisation is to reduce the relevant fault propagation scope and then reduce the relevant fault diagnosis scope. We develop a useful *stepwise scope reduction process* to reduce the relevant scope from both boundary directions towards the intermediate location of the target fault under diagnosis. We introduce the following two testing strategies for stepwise scope reduction:

(a) *The upper-boundary scope reduction strategy*

This stepwise scope reduction strategy aims to stepwise *reduce the upper boundary* of the relevant fault propagation scope and fault diagnosis scope from the upper boundary point towards the possible location of the target fault under diagnosis. A *key testing guideline* for effective fault diagnosis and localisation is to insert appropriate test contracts at certain selected testing points before the last upper boundary point and in the reverse direction of fault propagation in the relevant (sequential) execution path (as illustrated in Figure 7.3).

(b) *The lower-boundary scope reduction strategy*

Similarly, this stepwise scope reduction strategy aims to stepwise *increase the lower boundary* of the relevant fault propagation scope and fault diagnosis scope from the lower boundary point towards the possible location of the target fault under diagnosis. A *key testing guideline* is to insert appropriate test contracts at certain selected testing points after the last lower boundary point and in the same direction of fault propagation in the relevant (sequential) execution path (as illustrated in Figure 7.3).

(c) Guide to the use of the stepwise scope reduction strategies

These two testing strategies enable stepwise scope reduction from both upper and lower boundary directions towards the intermediate location of the fault under diagnosis. In practice, the tester can first apply the upper-boundary reduction strategy to conduct the *upper-boundary scope reduction process*, and then apply the lower-boundary reduction strategy to conduct the *lower-boundary scope reduction process*, or conduct joint *dual-direction boundary scope reduction* alternatively, depending on the actual testing circumstances and/or needs.

The stepwise scope reduction process is a major part of the CBFDD guidelines, and plays a key role in actual fault diagnosis and localisation. We can now further detail the main steps and associated technical aspects to illustrate how to apply the above two stepwise scope reduction processes for fault diagnosis and localisation.

(3.1) Step 3.1: *The upper-boundary scope reduction process*

Step 3.1.1: To realise the upper-boundary scope reduction strategy, we can insert an appropriate test contract to raise related warnings or exceptions at a selected testing point before the last upper boundary point (which may initially be at the final failure output point) to stop the development of fault propagation. If the inserted test contract can stop the fault propagation development at the selected testing point, this test contract is regarded as effective for scope reduction.

Step 3.1.2: The stopping point of the fault propagation development is where the inserted test contract is *violated*. This indicates that the relevant scope reduction can be carried out by reducing the relevant upper boundary to the new contract-violated point, which becomes the newly-reduced upper boundary. As the result of scope reduction, the new fault propagation scope now ranges only from the execution starting point to the new upper boundary point, and thus is smaller than the initial (maximum) fault propagation scope ranging from the execution starting point to the final failure output point.

Step 3.1.3: Accordingly, this new localised scope is the basis for producing the relevant newly-reduced fault diagnosis scope, which covers a range from the execution starting point to the new contract-violated point (which becomes the newly-reduced upper boundary). Therefore, the new fault diagnosis scope covers the relevant new fault propagation scope, and is reduced as the relevant new fault propagation scope becomes smaller.

Step 3.1.4: Following a similar stepwise process (Steps 3.1.1 to 3.1.3 as above) for further scope reduction, we can insert the same or a new test contract at a newly selected testing point in the reverse direction of fault propagation in the execution path before the last upper boundary point (i.e. before the last contract-violated point). Consequently, we can further reduce the fault propagation scope to a smaller scope ranging from the execution starting point to the new contract-violated point (which becomes the newly-reduced upper boundary). Accordingly, the relevant new fault diagnosis scope can also be reduced to a further localised scope and covers the

relevant newly-reduced fault propagation scope. The upper-boundary scope reduction process can be undertaken iteratively and incrementally as described above for further stepwise scope reduction.

(3.2) Step 3.2: *The lower-boundary scope reduction process*

The lower-boundary scope reduction process is similar to the above upper-boundary scope reduction process, but reduces the relevant scope by increasing the lower boundary towards the possible location of the target fault under diagnosis. A major difference is that we conduct stepwise scope reduction by inserting new test contracts at certain selected testing points towards the same direction of fault propagation in the execution path and after the last lower boundary point. The lower-boundary scope reduction process can be also undertaken iteratively and incrementally for further stepwise scope reduction.

(3.3) *Main advantages of the stepwise scope reduction process*

We can observe that the reduction of the relevant fault propagation scope and fault diagnosis scope can optimise fault diagnosis and localisation. One major advantage is that we now need to focus fault diagnosis and localisation only on the software execution part in the newly-reduced fault diagnosis scope, where there is a high probability of occurrence of the target fault under diagnosis. On the other hand, it is generally not necessary to examine the diagnosis-irrelevant range that is outside the newly-reduced fault diagnosis scope, where it has a low or almost no possibility of the fault occurring. Such a diagnosis-irrelevant range may be the software execution part between the newly-reduced upper boundary point and the initial upper boundary point (which may initially be at the final failure output point) in the case of the upper-boundary scope reduction. Or, such a diagnosis-irrelevant range may be the software execution part between the initial lower boundary point (which may initially be at the execution start point) and the newly-increased lower point in the case of the lower-boundary scope reduction. Therefore, the use of the stepwise scope reduction process can significantly improve fault diagnosis efficiency and reduce testing costs.

(4) Step #4: *Reducing the fault diagnosis scope to class/operation scope*

When the relevant fault diagnosis scope is constrained and reduced to a specific component unit (e.g. a class of the CUT), the complexity of fault diagnosis and localisation has also been reduced. Then, in the smaller scope of the component class, we can further apply a similar stepwise scope reduction process to fault diagnosis and localisation. By inserting appropriate test contracts in the component unit, it is possible to further reduce the relevant fault diagnosis scope to the much smaller scope of some closely-related class operation(s), which would be the minimum possible fault diagnosis scope.

(5) Step #5: *Locating the target fault that has been detected during testing*

When CBFDD reaches the above Step #4, it becomes much easier to diagnose and locate the actual position of the target fault for correction and removal to meet the particular testing objective.

(6) Step #6: *Correcting and removing the detected fault*

It is not always an easy task to correct and remove a specific component fault even though the fault has been detected and located. Clearly, fault correction and removal must follow component requirements and specifications. Although there is no general method or solution, we can still develop certain useful guidelines that can be applied to some particular situations for fault correction and removal.

For example, when the detected fault is located within the localised scope of a relevant operation, we can carry out fault correction and removal based on the following four possible fault cases:

(a) If the relevant operation is present, but the CUT does not actually execute the relevant operation as required, we need to change the relevant execution scenario by putting this operation in its correct execution path, so that this operation is executed at its correct execution point in the execution path of this CUT.

(b) If the relevant operation is present and is executed at its correct execution point, but its execution is incorrect or fails, for example, due to some incorrect invocation/usage of this operation (e.g. incorrect operation name use, incorrect operation parameters passing). In this case, we need to correctly invoke and use this operation at its correct execution point.

(c) If the relevant operation is present and is executed at its correct execution point, but its execution is incorrect or fails, for example, due to the incorrect definition/implementation of this operation in its home class (which consequently causes the incorrect operation execution return/result). In this case, we need to change its home class to correct the definition and/or implementation of this operation.

(d) If the relevant operation is not present or the CUT does not actually contain the relevant operation as required, we need to define this operation in its home class and/or include this operation at its correct execution point in this CUT.

(7) Guidelines remarks: *Interactive/incremental fault diagnosis and localisation*

With the CBFDD guidelines, Steps #1 to #6 can be undertaken iteratively and incrementally in actual fault diagnosis and localisation. In particular, the stepwise scope reduction process (e.g. Steps 3.1.1 to 3.1.4) needs to follow an iterative process to gradually reduce the rele-

vant fault propagation scope and fault diagnosis scope. Fault diagnosis and localisation need to follow an incremental process when additional test contracts are needed with CBCTD to diagnose and locate a specific target component fault (as illustrated in Figure 7.2).

7.6 Applying the CBFDD Method

As described in Sections 7.3 to 7.5, the CBFDD method enhances the TbC technique to realise the CfD goal for effective fault diagnosis and localisation by using a number of useful technical components, including the CBFDD process, the three fault diagnosis properties, the two stepwise upper/lower-boundary scope reduction strategies and the two associated processes, and the CBFDD guidelines. This section moves on to apply the CBFDD method. We employ the CPS case study to illustrate how to put the CBFDD method into practice to detect, diagnose and locate component faults.

7.6.1 Applying the CBFDD Process

As described earlier in Section 6.5, CBCTD can create a test group composed of related test operations and test contracts to examine a possible fault case scenario and uncover potential component faults. In this section, to undertake CIT for the CPS system, we follow the CBFDD process (as described in Section 7.4) to illustrate how to detect and diagnose possible component faults in the context of the CPS TUC1 integration testing scenario (as illustrated earlier in Figure 5.4, and as described earlier in Section 5.5.2 and Section 6.5). We use a particular CBCTD involving a basic test group 2.3 TG that consists of test operations 2.2 TO `goTo( gopace-cross-inPC, int )` and 2.3 TO `occupy()`, and test contract 2.3 ETC `checkState( inPhotoCell, "IN_PC_OCCUPIED" )` in the CPS TUC1 test sequence (as shown in Figure 7.4), in order to examine a fault case scenario of the in-PhotoCell sensor device in the CBFDD process.

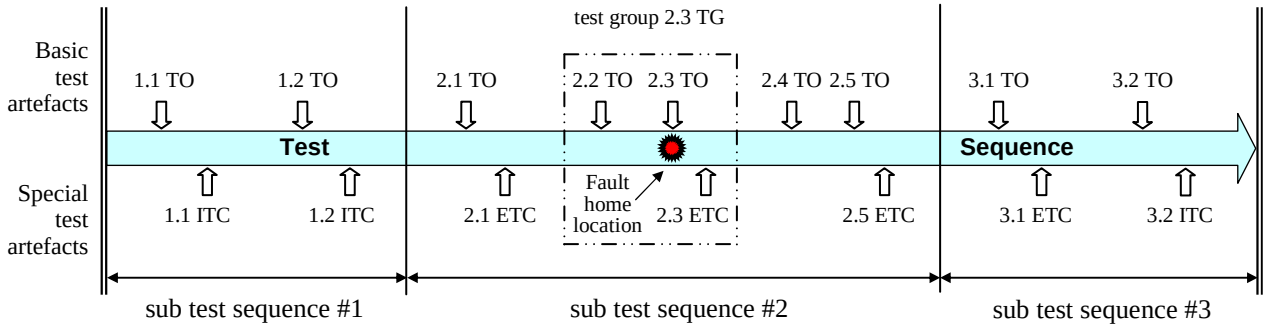


Figure 7.4 CBFDD: Fault Detection and Diagnosis (CPS TUC1 Test Sequence)

(1) *Fault case scenario*

If test contract 2.3 ETC in the current CBCTD returns *false*, a fault case scenario occurs: the in-PhotoCell sensor device is not in the correct state of "IN_PC_OCCUPIED" as expected, after test operation 2.3 TO has been performed. This means that this device fails to sense that the PAL entry point has been occupied by the entering car (i.e. this device fails to sense that the test car is accessing the PAL entry point). The failure output may show that this device still remains in the incorrect state of "IN_PC_CLEARED" or another unexpected state.

(2) *Fault consequence*

This fault case scenario may cause the PAL entry point not to be occupied as expected, and some subsequent operation (e.g. test operation 2.5 TO `clear`()) may not be executed as needed in the expected execution sequence of parking control operations, which may further lead to the entire CPS operation being halted at this failure output point.

(3) *Fault causes and analysis*

This fault case scenario indicates that the execution of test operation 2.3 TO fails in the CPS TUC1 integration testing context possibly with two main causes:

(i) Fault cause #1: the incorrect invocation/usage of test operation 2.3 TO; or

(ii) Fault cause #2: the incorrect definition/implementation of test operation 2.3 TO.

Note that either of these fault causes is related to test operation 2.3 TO and is examined with the current CBCTD in the current CPS TUC1 integration testing context.

(4) *Fault location*

Based on the above examination of fault causes, we can identify possible locations of the fault under diagnosis as follows:

(a) For the above fault cause #1, the fault most likely occurs in the caller class `CarController` in the car control component, where this integration class incorrectly invokes and/or uses test operation 2.3 TO in the integration context.

(b) For the above fault cause #2, the fault most likely occurs in its home class `PhotoCell`, where test operation 2.3 TO is incorrectly defined and/or implemented.

(5) *Fault-related test level*

The above two different fault locations by their nature indicate that the possible fault occurrence is related to the following two different component test levels:

(a) The fault location in (4) (a) above (which is related to the above fault cause #1) indicates that the fault occurrence is clearly pertinent to *inter-component integration testing*. This is because the fault is produced when the integration class `CarController` in the car

control component incorrectly invokes and/or uses operation occupy() of device class PhotoCell in the device control component, and the invocation is a typical object interaction to realise a component collaboration between the two CPS components. Accordingly, the above CBFDD activity shows that the current CBCTD (which involves a basic test group 2.3 TG) is able to examine and uncover a possible component fault related to CIT in the SCI context of the CPS system.

(b) The fault location in (4) (b) above (which is related to the above fault cause #2) indicates that the fault occurrence is clearly pertinent to *class unit testing*. This is because the fault is produced when operation occupy() is incorrectly defined/implemented inside its home class PhotoCell, which means that there may be an actual physical hardware fault of the in-PhotoCell sensor device. Accordingly, the above CBFDD activity shows the current CBCTD (which involves a basic test group 2.3 TG) can examine and uncover a possible component fault related to the component unit context of device class PhotoCell. This further indicates that the previous unit testing of this CPS device class might not turn out to be sufficiently adequate when the testing proceeds to the higher-level CIT of the CPS system.

The above illustrative example shows how the CBFDD process is applied to detect and diagnose actual component faults that are related to component integration testing and/or component unit testing. Following the CBFDD method, after the detected/located component fault (e.g. the above fault related to operation occupy()) is corrected and removed, we then need to conduct the appropriate integration/unit-level regression testing.

7.6.2 Diagnosing and Locating Target Component Faults

A key objective of the CBFDD method aims to apply a smaller number of well-designed test contracts (ITCs and ETCs) that can diagnose and locate a specific target component fault for a particular testing objective, in terms of low-overhead test contract coverage and desired testing effectiveness and efficiency (as described in Sections 7.5.4 and 7.5.5). To realise the above CfD goal, the CBFDD method provides a set of useful FDD guidelines, which are supported with the relevant fault diagnosis properties, the stepwise scope reduction strategies and processes (as described in Section 7.5). In this section, we employ some selected examples from the CPS case study to illustrate how to apply the CBFDD guidelines to diagnose and locate a specific target fault against the particular testing objective as follows: the CPS system must conform to the CPS special testing requirement #1 for the mandatory parking access safety rule – "*one access at a time*" (as described in Section B.2 in Appendix B).

7.6.2.1 A Specific Target Fault

Suppose that the CPS system encounters the following actual *major fault/failure scenario of the* CPS safety rule: while the current car enters the PAL entry point and is accessing the PAL but has not finished its complete PAL access yet, another unauthorised car illegally enters and accesses the PAL at the same time. This resulting failure is a safety violation of the "*one access at a time*" rule against the CPS special testing requirement #1. If the related fault is not corrected and removed immediately, a worse case scenario would be that two or more cars might access the PAL simultaneously, which could lead to hazardous collisions between cars in the CPS system.

7.6.2.2 Diagnosing and Locating the Specific Target Fault

Our FDD task is to diagnose and locate the specific target fault that causes the occurrence of this CPS operation failure and safety violation, and CBFDD activities start with analysing the above actual CPS safety rule failure scenario to seek and develop certain useful fault diagnostic solutions. Due to the nature of the occurrence of the actual CPS safety rule failure scenario, we need to apply the CBFDD method to diagnose and locate this specific target fault in two major possible testing contexts. In particular, we undertake CBFDD in the CPS system development environment as the current testing context (see the following sub Sections 7.6.2.2.1 and 7.6.2.2.2), and we also undertake CBFDD in the CPS user operational environment as the current testing context (see the next Section 7.6.2.3).

7.6.2.2.1 A Direct Fault Diagnosis Scenario Analysis

In this section, we undertake CBFDD based on the CPS system's requirements and design specifications, and examine the CPS safety rule failure scenario by conducting a direct fault diagnosis scenario analysis in the CPS system development environment as the current testing context. According to the CPS design, the traffic light device in the CPS system is responsible for authorising and controlling a car to enter and access the PAL, by using its two main control operations `setGreen`() and `setRed`() of class `TrafficLight` in the CPS system's device control component. The traffic light device with these two operations functions in the CPS system as follows:

(1) Operation `setGreen`() sets the traffic light device to the control state of "TL_GREEN" (i.e. the traffic light device turns to the GREEN signal), so that the next waiting car is allowed to enter the PAL.

(2) Operation `setRed`() sets the traffic light device to the control state of "TL_RED" (i.e. the traffic light device turns to the RED signal), so that the next waiting car is not allowed to enter the PAL and must wait for access permission (i.e. the car waits for the traffic light device to change to the GREEN signal).

(3) The traffic light device by design should maintain a basic CPS *control state consistency* feature as follows:

(a) The traffic light device should consistently remain in the control state of "TL_GREEN" after the successful execution of operation `setGreen`() and before the next execution of operation `setRed`().

(b) Similarly, the traffic light device should consistently remain in the control state of "TL_RED" after the successful execution of operation `setRed`() and before the next execution of operation `setGreen`().

(c) The traffic light device should shift its control state only from "TL_GREEN" to "TL_RED" and vice versa, and there should be no any other valid control state for the traffic light device at any time in the CPS system.

(d) Any other CPS operations should have no effect on the control state of the traffic light device at any time.

(Note that, for simplicity in the CPS system, here we do not consider the intermediate transitional signal of AMBER, when the traffic light device changes from the GREEN signal to the RED signal.)

The CPS safety rule failure scenario indicates that, after the current test car enters the PAL entry point, the traffic light device is not correctly set to the required control state of "TL_RED", which then causes the CPS system to fail in preventing another test car unexpectedly entering the PAL while the current test car is still accessing the PAL. When this failure scenario occurs, we can infer how the CPS operation unexpectedly fails in the CPS TUC1 integration context, in terms of the following possible *CPS fault cases* for the purpose of fault diagnosis and correction (which corresponds to Step #6 in the CBFDD guidelines as described in Section 7.5.5):

(a) The current CPS program may not actually execute operation `setRed`(), e.g. due to some incorrect execution path; or

(b) The execution of this operation is incorrect or fails, e.g. due to the incorrect invocation/usage of this operation; or

(c) The execution of this operation is incorrect or fails, e.g. due to the incorrect definition/implementation of this operation.

(d) The current CPS program may not actually contain operation `setRed`() in the execution path.

With any of these specific fault cases, a fault (namely FAULT_TL_RED) related to operation `setRed`() of the traffic light device is mistakenly produced, and because it is activated in the CPS TUC1 integration context, this fault eventually causes the occurrence of the actual CPS safety rule failure scenario. Therefore, our FDD task is to diagnose and locate this specific target FAULT_TL_RED fault, which is a component fault related to operation `setRed`() of class `TrafficLight` in the CPS system's device control component.

7.6.2.2.2 A Direct Fault Diagnostic Solution

The above direct fault diagnosis scenario analysis (as described in Section 7.6.2.2.1) can aid in developing certain useful testing solutions to conduct fault diagnosis and localisation. In the CPS system, the correct operation and use of the traffic light device is critical to support and realise the "one access at a time" rule. Accordingly, the CPS TUC1 test scenario needs to correctly use test operation 1.2 TO `setGreen`() to authorise a car to enter the PAL. When the car has just entered the PAL, the CPS TUC1 needs to correctly use test operation 3.2 TO `setRed`() to promptly disallow the next waiting car from entering the PAL while the current test car is accessing the PAL. Therefore, the above fault scenario is closely related to the traffic light device, especially to its control operation `setRed`() in the CPS TUC1 integration testing context.

According to the CPS design, the other two test scenarios CPS TUC2 and CPS TUC3 by their nature are not functionally responsible for controlling any operation of the traffic light device. This implies that the concealed FAULT_TL_RED fault, when it is activated in the CPS TUC1, could propagate from the CPS TUC1 to the CPS TUC2 and even to the CPS TUC3. If the exact failure output point is unknown, the last execution point of the CPS TUC3 may become the *final failure output point* in the worst-case situation (e.g. the entire system fails just at its last execution point), as described in Section 7.5.1. Consequently, the fault propagation development may extend across the entire parking cycle process covering all the three CPS test scenarios, which forms a typical (maximum) fault propagation scope for the specific target FAULT_TL_RED fault under diagnosis. Eventually, this concealed fault causes the occurrence of the CPS operation failure and safety violation (as described in Section 7.6.2.1).

Based on the above discussions, we can develop and obtain a direct fault diagnostic solution to identify and uncover this specific FAULT_TL_RED fault. The relevant CBCTD for the CPS TUC1 test scenario must correctly incorporate the two test operations `setGreen`() and `setRed`() of class `TrafficLight`, and their associated test contracts to examine and diagnose their execution, invocation/usage, and/or definition.

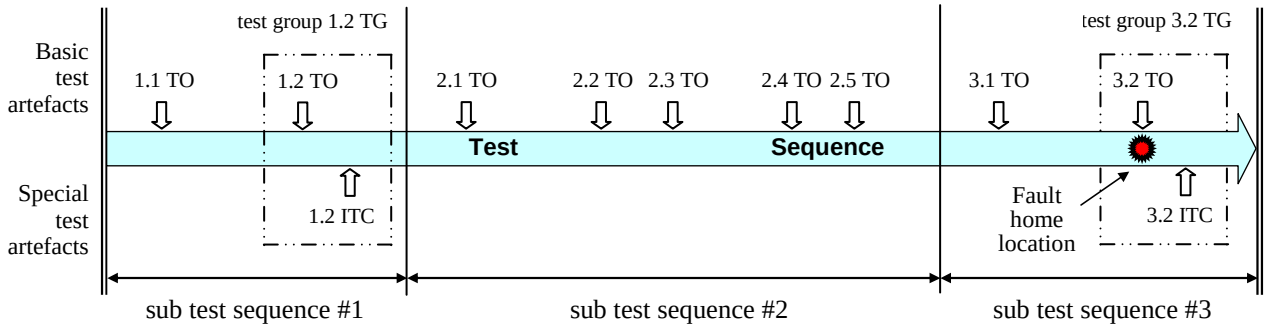


Figure 7.5 CBFDD: Fault Diagnosis and Localisation (CPS TUC1 Test Sequence)

In particular, the relevant CBCTD must correctly incorporate the following two basic test groups in the CPS TUC1 test sequence (as shown in Figure 7.5), which has the following diagnostic functions for the CIT purpose:

(1) One basic test group 1.2 TG contains test operation 1.2 TO `setGreen`() and its associated test contract 1.2 ITC `checkState`(`trafficLight`, "TL_GREEN").

This test contract is positioned at the selected testing point just after this test operation in the CPS TUC1 test sequence. This test group has the following diagnostic function: this test contract examines whether the traffic light device is in the correct control state of "TL_GREEN" as expected, just after test operation 1.2 TO `setGreen`() is performed. If and only if this test contract returns *true*, the next waiting car is allowed to enter the PAL.

(2) Another basic test group 3.2 TG contains test operation 3.2 TO `setRed`() and its associated test contract 3.2 ITC `checkState`(`trafficLight`, "TL_RED").

This test contract is positioned at the selected testing point just after this test operation in the CPS TUC1 test sequence. This test group has the following diagnostic function: this test contract examines whether the traffic light device is in the correct control state of "TL_RED" as expected, just after test operation 3.2 TO `setRed`() is performed. If and only if this test contract returns *true*, the next waiting car is disallowed from entering the PAL and must wait for access permission.

(3) Applying the basic CPS control state consistency feature (as described in Section 7.6.2.2.1)

In addition, our FDD makes use of the related basic CPS control state consistency feature for effective fault diagnosis. When test contract 1.2 ITC in basic test group 1.2 TG returns *true* (i.e. test operation 1.2 TO `setGreen`() is executed correctly), the traffic light device will remain in the control state of "TL_GREEN" until the next execution point of test operation 3.2

TO `setRed`(). Any other CPS operations (e.g. test operations 2.1 TO, 2.2 TO, 2.3 TO, 2.4 TO, 2.5 TO and 3.1 TO, as shown in Figure 7.5) should have no effect on the control state of the traffic light device at any time (as described in Section 7.6.2.2.1). Accordingly, this can bring out an important testing advantage that it is not needed to check the state of the traffic light device between the successful execution of operation `setGreen`() and the next execution of operation `setRed`(), which can reduce the requirement for a number of test contracts that can improve testing efficiency and performance.

(4) Dual testing roles (also as described earlier in Section 6.5.1.1)

Based on the related basic CPS control state consistency feature, test contract 1.2 ITC actually acts in *dual testing roles* to facilitate fault diagnosis. This test contract works as a post-condition assertion of test operation 1.2 TO `setGreen`() in basic test group 1.2 TG, and also as an additional precondition assertion of test operation 3.2 TO `setRed`() in basic test group 3.2 TG, even though this test contract is not positioned just before test operation 3.2 TO. One testing advantage of such an additional precondition attribute is to ensure that the traffic light device shifts its control state only from "TL_GREEN" to "TL_RED" and vice versa, and there is no third valid control state for the traffic light device at any time in the CPS system (as described in Section 7.6.2.2.1).

Accordingly, we can see that component tests with the above CBCTD are able to detect, diagnose and locate the specific FAULT_TL_RED fault in the CPS TUC1 integration testing context. Moreover, this CBCTD only needs fewer test contracts (e.g. two selected test contracts at two selected testing points in this situation) to fulfil this specific fault diagnosis task. The illustrative example above demonstrates that our CBFDD method is capable of achieving the CfD goal in terms of low-overhead test contract coverage and desired testing efficiency and performance, compared with the TbC test contract criteria based FDD approach (as described in Section 7.5.3).

Note that the above direct fault diagnostic solution implicitly depends on some *testing-related assumptions* as follows:

(a) The tester is able to access component requirements and/or design specifications.

(b) The tester is able to undertake critical fault diagnosis scenario analysis, such as the direct fault diagnosis scenario analysis undertaken in the CPS system development environment as the current testing context (as described in Section 7.6.2.2.1 above).

(c) The tester is able to obtain and make use of certain testing-support features, such as the basic CPS control state consistency feature designed for the traffic light device in the CPS system (as described in Section 7.6.2.2.1 above).

The above testing-related assumptions are typically applicable to the testers on the component development/production side, who have certain testing advantages compared with the testers on the component user side (as described earlier in Section 2.3.4). Technically, this direct fault diagnostic solution can significantly simplify the steps from the CBFDD guidelines applied to diagnose and locate the specific target FAULT_TL_RED fault, which are outlined as follows:

(1) The actual CPS safety rule failure scenario occurs in the CPS user operational environment, which is a system integration context. Accordingly, the testing is at the level of integration testing, which also covers Step #1 in the CBFDD guidelines.

(2) The direct fault diagnosis scenario analysis indicates that the testing can be conducted in the CPS TUC1 test scenario context (as described in Section 7.6.2.2.1). Accordingly, we can use the CPS TUC1 test scenario as the basic fault diagnosis scope to delimit and constrain the relevant fault propagation scope. This means that we only need to conduct CBFDD within the range of the CPS TUC1 test scenario, even though the relevant fault propagation scope may spread out to the entire parking cycle process covering all the three CPS test scenarios (as described above). This simplifies the stepwise scope reduction, and also covers Step #2 and Step #3 in the CBFDD guidelines.

(3) The direct fault diagnosis scenario analysis indicates that the specific FAULT_TL_RED fault is related to operation `setRed`() of the traffic light device (as described in Section 7.6.2.2.1). This simplifies the fault diagnosis process, and also covers Step #4 and Step #5 in the CBFDD guidelines.

(4) The CPS fault cases identified with the direct fault diagnosis scenario analysis (as described in Section 7.6.2.2.1) facilitate correcting and removing the specific FAULT_TL_RED fault. This covers Step #6 in the CBFDD guidelines.

7.6.2.3 Stepwise Diagnosis and Localisation of the Specific Target Fault

The direct fault diagnostic solution (as described in Section 7.6.2.2.2 above) is not always attainable or available in testing practice, due to the uncertain complexity of software component and systems under test. On the other hand, the above testing-related assumptions (as described in Section 7.6.2.2.2 above) are not applicable in all testing situations. For example, most testers on the component user side usually would not have the privilege of accessing the full information of component requirements and design specifications (as described earlier in Section 2.3.4). On the other hand, the actual fault diagnosis may not always be able to obtain and/or make use

of certain testing-support features (e.g. the basic CPS control state consistency feature as described in Section 7.6.2.2.1).

The CBFDD guidelines (as described in Section 7.5.5) are particularly applicable to the above situations, by applying all steps for stepwise fault diagnosis and localisation. This section illustrates all the steps with the CBFDD guidelines that are applied to develop and attain a stepwise fault diagnostic solution to diagnose and locate the specific target FAULT_TL_RED fault. A primary objective is to show how the CBFDD guidelines work and demonstrate the applicability of our CBFDD method.

7.6.2.3.1 Fault Diagnosis Scenario Analysis

To apply the CBFDD guidelines effectively, it is first necessary to conduct the relevant fault diagnosis scenario analysis. From the user perspective of the CPS system under test (e.g. concerning relevant system operational functions provided for the users), the tester can analyse the CPS safety rule failure scenario (as described in Section 7.6.2.1) that could occur in the CPS user operational environment as the current testing context as follows:

(1) The CPS system uses the traffic light device to authorise a car to enter and access the PAL. The safety violation of the "one access at a time" rule is due to the operational failure of the traffic light device, i.e. it fails in changing to the RED signal to prevent the next car from entering the PAL, while the current car is still accessing the PAL. Accordingly, our major FDD task is to diagnose and locate the specific target FAULT_TL_RED fault related to the traffic light device, which causes the CPS operation failure and safety violation (as described in Section 7.6.2.1).

(2) To diagnose and locate the specific target FAULT_TL_RED fault:

(a) The relevant CBCTD needs to examine and diagnose the related operation of the traffic light device (namely TO_TL_RED), which should turn to the RED signal to prevent the next car from entering the PAL, after the current car enters the PAL. Test operation TO_TL_RED functions equivalently to test operation 3.2 TO that is included in the CPS design and is shown in the basic test group 3.2 TG in Figure 7.5.

(b) For the fault diagnostic purpose as in (2) (a) above, the relevant CBCTD needs to design a crucial test contract (namely TC_TL_RED) that can examine and diagnose whether the traffic light device is currently in the correct control state of "TL_RED" as expected, after the current car enters the PAL entry point and when the current car is accessing the PAL. Test contract TC_TL_RED functions equivalently to test contract 3.2 ITC in the basic test group 3.2 TG (as shown in Figure 7.5).

(c) Combining (2) (a) and (b) above, the relevant FDD task needs to apply test contract TC_TL_RED to examine and diagnose test operation TO_TL_RED. With regard to related diagnostic functions, test contract TC_TL_RED needs to be verified after test operation TO_TL_RED is executed.

- If this test contract returns *false*, the execution of this test operation fails, which results in the CPS operation failure and safety violation (as described in Section 7.6.2.1). In this case, this test contract needs to raise relevant warnings or exceptions at its testing point to stop fault propagation development for the purpose of fault diagnosis and localisation.
- If this test contract returns *true*, the execution of this test operation is correct as expected to prevent the next car from entering the PAL, while the current car is still accessing the PAL. However, the current CPS system operation does not work correctly in this situation.

(d) However, it is not exactly known (at least at the initial testing stages) where test operation TO_TL_RED is in the CPS system under test. This may be due to certain testing-related factors in practice, for example, the nature of the uncertain component complexity and/or the limited information of component requirements and design specifications. Accordingly, this causes certain practical difficulties to select appropriate testing points and to apply test contract TC_TL_RED to effectively examine and diagnose the related test operation TO_TL_RED. Therefore, it is necessary to apply the diagnostic steps with the CBFDD guidelines to undertake our FDD task.

(3) To fulfil our major FDD task as described in (1) and (2) above, the relevant CBCTD needs to be able to conduct some supporting fault diagnosis to ensure normal CPS operation:

(a) The relevant CBCTD needs to examine and diagnose the related operation of the traffic light device (namely TO_TL_GREEN), which should turn to the GREEN signal to allow the next waiting car to enter the PAL for the normal CPS operation. Test operation TO_TL_GREEN functions equivalently to test operation 1.2 TO that is included in the CPS design and is shown in the basic test group 1.2 TG in Figure 7.5.

(b) For the fault diagnostic purpose as in (3) (a) above, the relevant CBCTD needs to design another necessary test contract (namely TC_TL_GREEN) that can examine and diagnose whether the traffic light device is currently in the correct control state of “TL_GREEN” as expected, which allows the next waiting car to enter the PAL for the normal CPS operation. Test contract TC_TL_GREEN functions equivalently to test contract 1.2 ITC in the basic test group 1.2 TG (as shown in Figure 7.5).

(c) Combining (3) (a) and (b) above, the supporting FDD task needs to apply test contract TC_TL_GREEN to examine and diagnose test operation TO_TL_GREEN. With regard to related diagnostic functions, test contract TC_TL_GREEN needs to be verified after test operation TO_TL_GREEN is executed.

- If this test contract returns *true*, the execution of this test operation is correct as expected. This means that the traffic light device correctly turns to the GREEN signal to allow the next waiting car to enter the PAL for normal CPS operation. The current CPS system operation works correctly in this situation.
- If this test contract returns *false*, the execution of this test operation fails to allow the next waiting car to enter the PAL for the normal CPS operation. In this case, this test contract needs to raise relevant warnings or exceptions at its testing point to stop fault propagation development for the purpose of fault diagnosis and localisation.

(d) However, it is not exactly known (at least at the initial testing stages) where test operation TO_TL_GREEN is in the CPS system under test, for example, due to the nature of the uncertain component complexity and/or the limited information of component requirements and design specifications. Accordingly, this causes certain practical difficulties to select appropriate testing points and to apply test contract TC_TL_GREEN to effectively examine and diagnose the related test operation TO_TL_GREEN. Therefore, it is necessary to apply the diagnostic steps with the CBFDD guidelines to conduct the above supporting FDD task.

7.6.2.3.2 Stepwise Fault Diagnosis and Localisation

Based on the above fault diagnosis scenario analysis (as described in Section 7.6.2.3.1), the following illustrates how to stepwise diagnose and locate the specific target FAULT_TL_RED fault (as illustrated in Figure 7.6), by using the two main test contracts TC_TL_RED and TC_TL_GREEN identified above. We apply the main technical steps and stepwise scope reduction process described in the CBFDD guidelines to undertake stepwise fault diagnosis and localisation, in conjunction with the features of the relevant fault diagnosis properties, the stepwise upper/lower-boundary scope reduction strategies and processes, which all were described in Section 7.5.

Note that in Figure 7.6, $TC_{1.0}$ ($TC_{2.0}$, $TC_{3.0}$) denotes the *first test contract* at the first testing point that is just before the first execution point in the CPS TUC1 (TUC2, TUC3) test scenario. Similarly, $TC_{1.L}$ ($TC_{2.L}$, $TC_{3.L}$) denotes the *last test contract* at the last testing point that is just after the last execution point in the CPS TUC1 (TUC2, TUC3) test scenario.

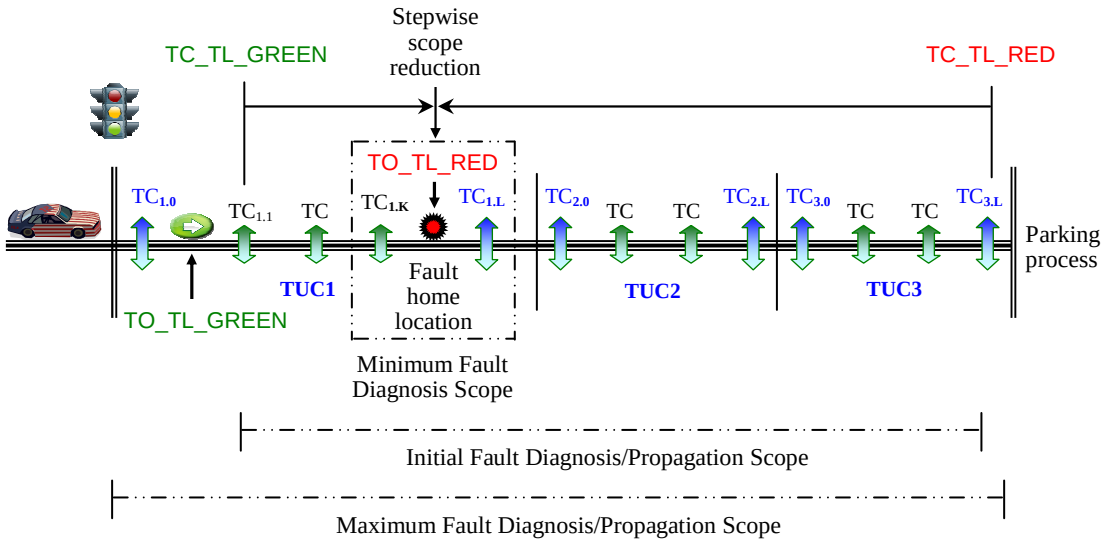


Figure 7.6 CBFDD: Stepwise Fault Diagnosis and Localisation

(1) Step #1: *Determining test levels: integration testing*

We first need to find the initial fault diagnosis scope to determine the relevant test level. Because the exact failure output point may actually be unknown, we need to consider the worst-case situation for the actual fault propagation scope and the actual fault diagnosis scope (as described in Sections 7.5.1 and 7.5.2). In the case of the CPS system under test, this means that the maximum fault propagation scope may range from the first execution point of the CPS TUC1 range to the last execution point of the CPS TUC3 range. Accordingly, the *maximum fault diagnosis scope* may range from the first testing point (at the $TC_{1.0}$ position) just before the first execution point of the CPS TUC1 range to the last testing point (at the $TC_{3.L}$ position) just after the last execution point of the CPS TUC3 range (as illustrated in Figure 7.6). This would be the worst-case situation for all possible faults in the CPS system.

For the specific target FAULT_TL_RED fault under diagnosis, we initially insert test contract TC_TL_GREEN at the $TC_{1.1}$ position in the CPS TUC1 range, and test contract TC_TL_RED at the $TC_{3.L}$ position in the CPS TUC3 range. The *initial fault diagnosis scope* ranges from the $TC_{1.1}$ position in the CPS TUC1 range to the $TC_{3.L}$ position in the CPS TUC3 range (as illustrated in Figure 7.6). Accordingly, we need to diagnose and locate this fault in the CPS integration context, which, in this case, covers the CPS TUC1, TUC2 and TUC3 range. Therefore, the task of diagnosing and locating this specific target fault is typically related to component integration testing. (Note that the $TC_{1.1}$ position is the first *valid testing point* for test contract TC_TL_GREEN and the $TC_{3.L}$ position is the last valid testing point for test contract TC_TL_RED. However, the $TC_{1.0}$ position is not a valid testing point for test contract TC_TL_GREEN, which is to be explained in Step 3.2 below.)

Because the actual CPS safety rule failure scenario occurs after the current test car enters

the PAL entry point, test contract TC_TL_RED at the last testing point (at the $TC_{3.L}$ position) incorrectly returns *false*. The violation of this test contract indicates that the traffic light device is currently NOT in the expected control state of "TL_RED", and so fails to prevent another car entering the PAL when the current test car is accessing the PAL. This means that the traffic light device currently has the FAULT_TL_RED fault, which occurs at some execution point before the last testing point at the $TC_{3.L}$ position in the CPS TUC3 range, even though, in the initial testing stages, we do not exactly know where this specific fault is in the CPS system under test.

(2) Step #2: *Determining the fault propagation direction for scope reduction*

By inserting test contract TC_TL_RED at a testing point before the last-diagnosed testing point (initially at the $TC_{3.L}$ position) in the CPS TUC3 range, we can ascertain the direction of fault propagation development related to the specific target FAULT_TL_RED fault. When the fault propagation development is stopped by this test contract before the final failure output point (initially at the last execution point), the direction of fault propagation development is determined, i.e. this fault propagates from its under-diagnosis home location to the CPS TUC1 range to the CPS TUC2 range to the CPS TUC3 range (as illustrated in Figure 7.6).

(3) Step #3: *Applying the stepwise scope reduction process to reduce the fault propagation scope and the fault diagnosis scope*

We apply the stepwise scope reduction process to diagnose and locate the specific target FAULT_TL_RED fault. The stepwise scope reduction process starts with the above initial fault diagnosis scope, where test contract TC_TL_GREEN at the $TC_{1.1}$ position acts as the lower boundary, and test contract TC_TL_RED at the $TC_{3.L}$ position acts as the upper boundary. We first apply the upper-boundary scope reduction process and then apply the lower-boundary scope reduction process (as described in Section 7.5.5).

(3.1) Step 3.1: *Applying the upper-boundary scope reduction process*

Following the upper-boundary scope reduction strategy (as described in Section 7.5.5), we apply the following *key testing guideline*: insert test contract TC_TL_RED at certain testing points in the CPS TUC3 range, before the last upper boundary point (initially it is at the last testing point at the $TC_{3.L}$ position) and towards the reverse direction of fault propagation development. We now conduct the stepwise process for upper-boundary scope reduction, by using test contract TC_TL_RED to reduce both the relevant fault propagation scope and the relevant fault diagnosis scope. Note that, during the following stepwise scope reduction process, we provisionally leave test contract TC_TL_GREEN at the lower boundary point unchanged (initially at the $TC_{1.1}$ position in the CPS TUC1 range).

(3.1.1) Stepwise reduction of the fault propagation scope and the fault diagnosis scope related

to the CPS TUC3 range (as illustrated in Figure 7.7)

With test contract TC_TL_RED being violated at a selected testing point in the CPS TUC3 range, we get the same occurrence of the actual CPS safety rule failure scenario related to the specific target FAULT_TL_RED fault. To diagnose and locate this fault, we are able to stop the fault propagation development by inserting this test contract at a new *selected testing point* before the last upper boundary point (initially at the $TC_{3.L}$ position) in the CPS TUC3 range. Accordingly, the current upper boundary of the relevant fault propagation scope is constrained and reduced to the new stopping point of fault propagation development, that is, the new upper boundary point is at the new selected testing point before the last upper boundary point in the CPS TUC3 range. This means that the relevant fault propagation scope is reduced to become a smaller localised range from the unchanged lower boundary point to the new upper boundary point. Consequently, the relevant fault diagnosis scope also is reduced and covers the current fault propagation scope (as illustrated in Figure 7.7).

After conducting a similar stepwise scope reduction process iteratively and incrementally by using the upper-boundary scope reduction strategy, we can obtain the smallest possible fault diagnosis scope related to the CPS TUC3 range, which ranges from the unchanged lower boundary point (initially at the $TC_{1.1}$ position) to the finally-reduced upper boundary point at the $TC_{3.0}$ position (as illustrated in Figure 7.7).

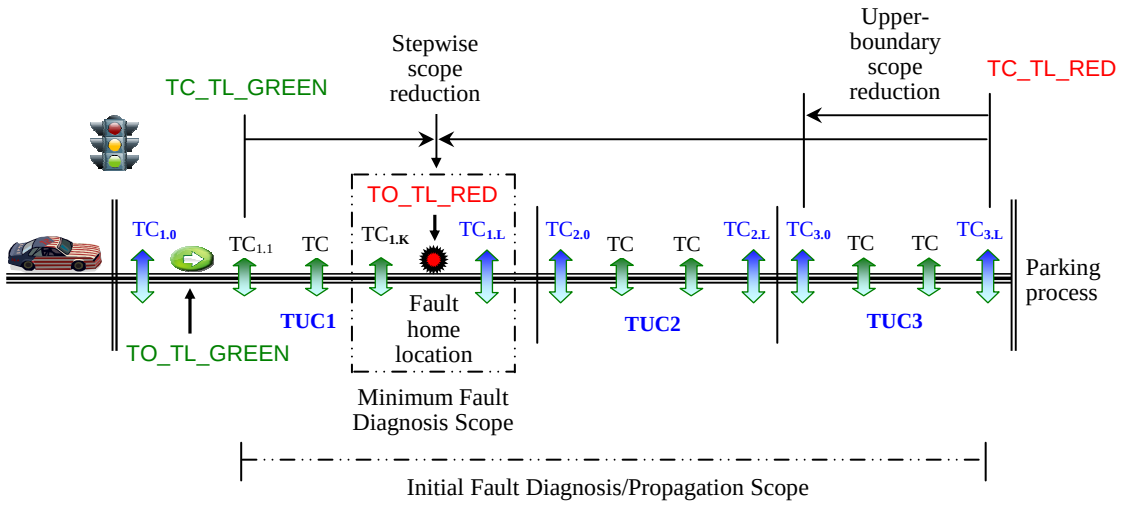


Figure 7.7 CBFDD: Stepwise Fault Diagnosis and Localisation (Step 3.1.1)

Within this newly-reduced fault diagnosis scope, because the $TC_{3.0}$ position is the first testing point before the first execution point in the CPS TUC3 integration context, we can reasonably exclude the possibility that the specific target FAULT_TL_RED fault may exist in the CPS TUC3 range. Therefore, applying this stepwise scope reduction process can greatly constrain the fault propagation scope and reduce the fault diagnosis scope to be smaller only within

the newly-reduced CPS integration context: in this case, the CPS TUC1 and TUC2 range. As the result of scope reduction, we can obtain the new stepwise-reduced fault diagnosis scope that ranges from the unchanged lower boundary point (initially at the $TC_{1.1}$ position) to the last testing point at the $TC_{2.L}$ position (as the new upper boundary point) in the CPS TUC2 range (as illustrated in Figure 7.8).

(3.1.2) Stepwise reducing the fault propagation scope and the fault diagnosis scope related to the CPS TUC2 range (as illustrated in Figure 7.8)

In the CPS TUC2 range, applying the upper-boundary scope reduction strategy to conduct a similar upper-boundary scope reduction process as described in Step 3.1.1 above, we can further reduce the upper boundary point from the $TC_{2.L}$ position to the $TC_{2.0}$ position, and obtain a further localised scope for fault propagation and fault diagnosis. Accordingly, we can obtain the smallest possible fault diagnosis scope related to the CPS TUC2 range, which ranges from the unchanged lower boundary point (initially at the $TC_{1.1}$ position) to the finally-reduced upper boundary point at the $TC_{2.0}$ position (as illustrated in Figure 7.8).

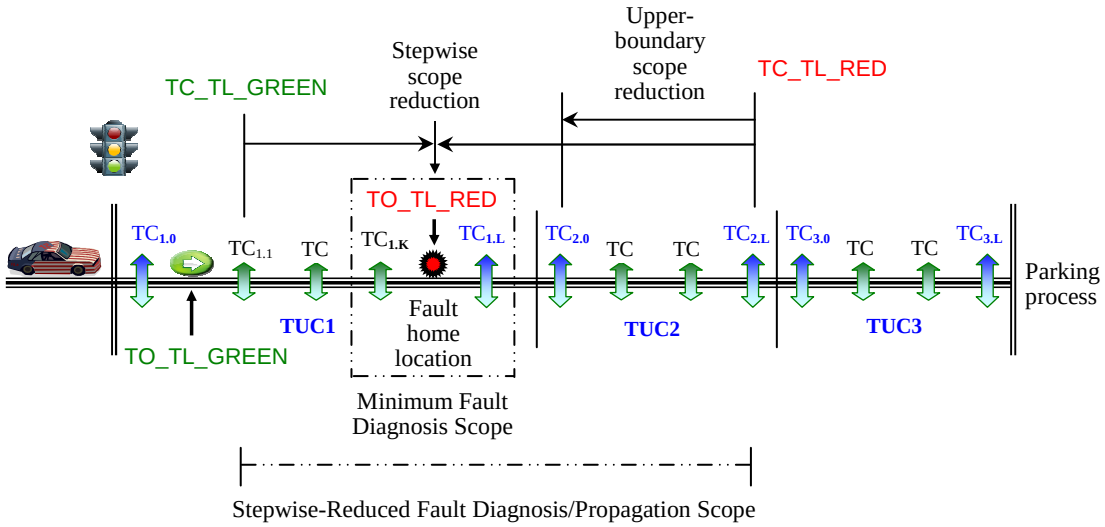


Figure 7.8 CBFDD: Stepwise Fault Diagnosis and Localisation (Step 3.1.2)

Similarly, in this newly-reduced fault diagnosis scope, because the $TC_{2.0}$ position is the first testing point before the first operation execution in the CPS TUC2 integration context, we can reasonably exclude the possibility that the specific target FAULT_TL_RED fault may exist in the CPS TUC2 range. This means that we can further constrain the fault propagation scope and reduce the fault diagnosis scope to be smaller only within the newly-reduced CPS integration context, i.e. the CPS TUC1 range. Therefore, as the result of further scope reduction, we can obtain the new stepwise-reduced fault diagnosis scope that ranges from the unchanged lower boundary point (initially at the $TC_{1.1}$ position) to the last testing point at the $TC_{1.L}$ position (as the new upper boundary point) in the CPS TUC1 range (as illustrated in Figure 7.9).

(3.1.3) Stepwise reducing the fault propagation scope and the fault diagnosis scope related to the CPS TUC1 range (as illustrated in Figure 7.9)

The fault diagnosis scenario analysis (as described in Section 7.6.2.3.1) indicates that it is difficult to select an appropriate testing point and to apply test contract TC_TL_RED to effectively examine and diagnose the related test operation TO_TL_RED, because it is not exactly known where this test operation is in the CPS system under test. A key objective of applying the stepwise scope reduction process is that reducing the relevant scope can considerably aid in finding the possible location of the related test operation TO_TL_RED in the CPS system, and an appropriate testing point for effectively applying test contract TC_TL_RED. The result of stepwise scope reduction conducted in Step 3.1.1 and Step 3.1.2 above shows that the related test operation TO_TL_RED does not exist in the CPS TUC3 and TUC2 range, which, in this case, matches the actual design of the CPS system. Accordingly, this implies that the relevant scope reduction must end up at some point within the CPS TUC1 range.

Now in the CPS TUC1 range, the current fault diagnosis scope covers the relevant fault propagation scope and ranges from the unchanged lower boundary point (initially at the $TC_{1.1}$ position) to the last testing point at the $TC_{1.L}$ position (as the upper boundary point), as illustrated in Figure 7.9. To further the upper-boundary scope reduction, we need to select a next contract testing point before the current upper boundary point. However, selecting a next contract testing point before the current upper boundary point at the $TC_{1.L}$ position leads to finding the location of the related test operation TO_TL_RED used in the CPS TUC1 test scenario, which, in this case, matches the actual design of the CPS system. This means that, for test contract TC_TL_RED, there is no valid testing point before the current upper boundary point at the $TC_{1.L}$ position in the CPS TUC1 range, because this test contract by its nature must be verified *after* this test operation is executed, based on the fault diagnosis scenario analysis (as described in Section 7.6.2.3.1). In other words, test contract TC_TL_RED by its nature has no effect on test operation TO_TL_RED if this test contract is positioned *before* this test operation. The fault-related test operation is now found at this execution point in the CPS TUC1 range. Accordingly, the current upper boundary point at the $TC_{1.L}$ position becomes the last *valid testing point* for this test contract, and because this testing point is positioned just after this test operation, it is the better-selected testing point for this test contract to effectively examine and diagnose this test operation.

Therefore, the relevant upper-boundary scope reduction process would reasonably end up at this last valid testing point in the CPS TUC1 range. The final stepwise-reduced fault diagnosis scope ranges from the unchanged lower boundary point (initially at the $TC_{1.1}$ position) to the last valid testing point at the $TC_{1.L}$ position (which becomes the final upper boundary point) in the CPS TUC1 range, as illustrated in Figure 7.10.

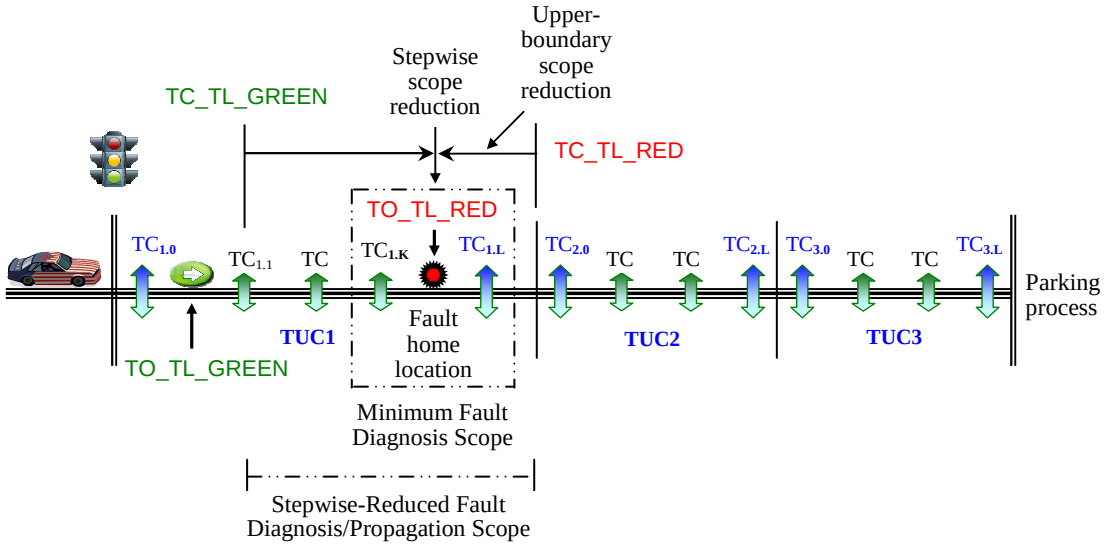


Figure 7.9 CBFDD: Stepwise Fault Diagnosis and Localisation (Step 3.1.3)

(3.2) Step 3.2: *Applying the lower-boundary scope reduction process*

The result of the upper-boundary scope reduction process conducted in Step 3.1 above (including Step 3.1.1 to Step 3.1.3) shows that test contract TC_TL_RED needs to be positioned at the $TC_{1.L}$ position as the final upper boundary point, and is verified as the postcondition assertion just after the execution of operation TO_TL_RED. For the purpose of rigorously diagnosing and locating the specific target FAULT_TL_RED, it is necessary to identify the final lower boundary point and obtain the final minimum fault diagnosis scope that ranges from the final lower boundary point to the above final upper boundary point. Moreover, it is also necessary to obtain the relevant test contract as the precondition assertion for effectively examining and diagnosing test operation TO_TL_RED in the final minimum fault diagnosis scope. Accordingly, we need to apply the lower-boundary scope reduction process.

Following the lower-boundary scope reduction strategy (as described in Section 7.5.5), we apply the following *key testing guideline*: inserting test contract TC_TL_GREEN at certain testing points in the CPS TUC1 range, after the last lower boundary point (initially it may be at the first testing point at the $TC_{1.0}$ position) and towards the same direction of fault propagation development. Note that, during the following stepwise scope reduction process, we maintain test contract TC_TL_RED at the above final upper boundary point unchanged (at the $TC_{1.L}$ position) in the CPS TUC1 range.

(3.2.1) Identifying the first valid testing point and the initial fault diagnosis scope

To conduct the lower-boundary scope reduction, we need to select a contract testing point after the first testing point at the $TC_{1.0}$ position in the CPS TUC1 range. However, selecting a contract testing point after the first testing point at the $TC_{1.0}$ position leads to finding the loca-

tion of the related test operation TO_TL_GREEN used in the CPS TUC1 test scenario, which, in this case, matches the actual design of the CPS system. This means that the $TC_{1.0}$ position is not a valid testing point for test contract TC_TL_GREEN (as indicated in Step #1 above), because this test contract by its nature must be verified *after* this test operation is executed, based on the fault diagnosis scenario analysis (as described in Section 7.6.2.3.1). In other words, test contract TC_TL_GREEN by its nature has no effect on test operation TO_TL_GREEN if this test contract is positioned *before* this test operation. Accordingly, the $TC_{1.1}$ position, which is positioned just after test operation TO_TL_GREEN, becomes the first *valid testing point* for test contract TC_TL_GREEN and the starting lower boundary point for stepwise scope reduction.

Therefore, the starting fault diagnosis scope for the lower-boundary scope reduction process is the same as the last stepwise-reduced fault diagnosis scope (which is resulted from Step 3.1 above). It covers the relevant fault propagation scope and ranges from the starting lower boundary point (at the $TC_{1.1}$ position) to the final upper boundary point (at the $TC_{1.L}$ position) in the CPS TUC1 range (as illustrated in Figure 7.10).

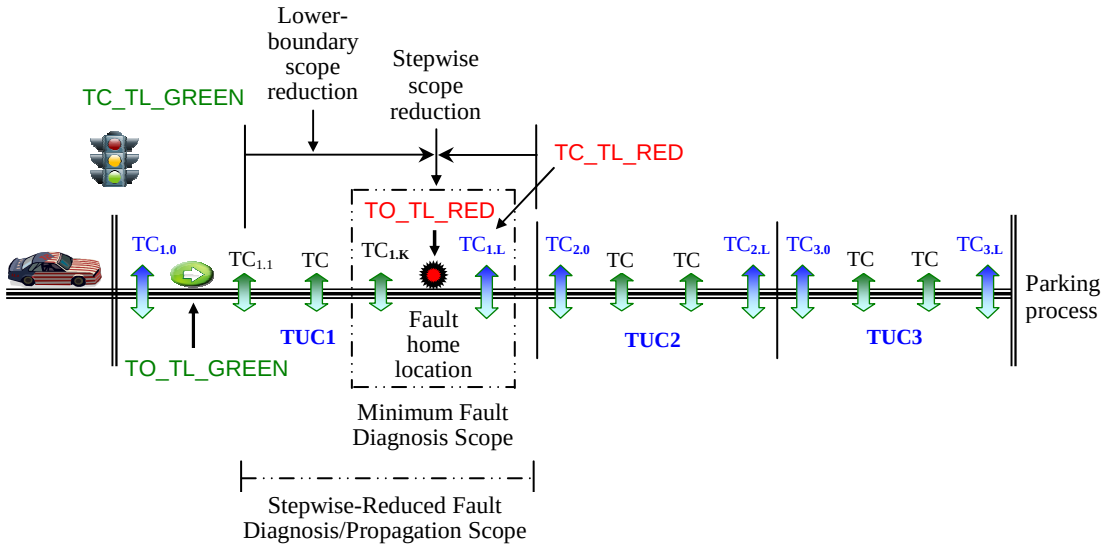


Figure 7.10 CBFDD: Stepwise Fault Diagnosis and Localisation (Step 3.2.1)

(3.2.2) Identifying the valid testing points and the valid testing range

We need to identify the valid testing points and obtain the valid testing range for test contract TC_TL_GREEN. Based on the fault diagnosis scenario analysis (as described in Section 7.6.2.3.1), test contract TC_TL_GREEN by its nature should be positioned at a contract testing point after the execution point of operation TO_TL_GREEN. In addition, test contract TC_TL_GREEN by its nature has no effect on test operation TO_TL_RED even if this test contract is positioned *after* this test operation in the CPS TUC1 range. Accordingly, a *valid testing point* for test contract TC_TL_GREEN is after the execution point of operation TO_TL_GREEN, towards the same direction of fault propagation development, and before the

execution point of operation TO_TL_RED. In particular, the first valid testing point is at the $TC_{1.1}$ position, and the last valid testing point is at the $TC_{1.K}$ position that is just before the execution point of operation TO_TL_RED in the CPS TUC1 range. These valid testing points form the *valid testing range* for test contract TC_TL_GREEN, which ranges from the first valid testing point at the $TC_{1.1}$ position to the last valid testing point at the $TC_{1.K}$ position in the CPS TUC1 range (as illustrated in Figure 7.10). Therefore, test contract TC_TL_GREEN must be applied and verified at a valid testing point within this valid testing range.

(3.2.3) Stepwise reducing the fault diagnosis/propagation scope

The fault diagnosis scenario analysis (as described in Section 7.6.2.3.1) indicates that the traffic light device currently shows the GREEN signal to allow the current waiting car to enter the PAL for the normal CPS operation. This means that the execution of operation TO_TL_GREEN is correct and test contract TC_TL_GREEN verified at the $TC_{1.1}$ position returns *true*. From the perspective of the CPS operational functions in the CPS user operational environment, the CPS system should maintain this correct CPS operational status in the execution range from the execution starting point where the current car starts entering the PAL entry point to the execution ending point where the current car finishes entering the PAL entry point. Then, just after the current car finishes entering the PAL entry point, the CPS system should immediately change the traffic light device from the GREEN signal to the RED signal to prevent the next car from entering the PAL, while the current car is accessing the PAL. In other words, from the perspective of the CPS operational functions, the CPS system should maintain this correct CPS operational status for the traffic light device in the execution range that is after the execution point of operation TO_TL_GREEN and before the execution point of operation TO_TL_RED. This property of the CPS system is equivalent to the basic CPS control state consistency feature as described in Section 7.6.2.2.1.

The abovementioned CPS operational status can be examined with test contract TC_TL_GREEN in conjunction with applying the lower-boundary scope reduction process. By inserting test contract TC_TL_GREEN at a new selected testing point after the starting lower boundary point (at the $TC_{1.1}$ position) and in the above valid testing range, the testing shows that this test contract by its nature returns *true*, which, in this case, matches the actual design of the CPS system. Accordingly, the new lower boundary point can be increased to this new selected testing point after the starting lower boundary point (at the $TC_{1.1}$ position), and the new fault diagnosis scope is reduced to be a smaller range from this newly-increased lower boundary point to the above final upper boundary point (at the $TC_{1.L}$ position) in the CPS TUC1 range.

After conducting a similar stepwise scope reduction process iteratively and incrementally by using the lower-boundary scope reduction strategy, we can obtain the final (and increased) lower boundary point at the $TC_{1.K}$ position just before the execution point of operation

TO_TL_RED in the CPS TUC1 range. Test contract TC_TL_GREEN verified at the $TC_{1.K}$ position returns *true*, confirming that the abovementioned CPS operational status is maintained consistently. Because the $TC_{1.K}$ position is the last valid testing point in the above valid testing range for this test contract, the relevant lower-boundary scope reduction process would reasonably end up at this finally-increased lower boundary point at the $TC_{1.K}$ position in the CPS TUC1 range. Accordingly, the final stepwise-reduced fault diagnosis scope ranges from the testing point at the $TC_{1.K}$ position (which becomes the final lower boundary point) to the above final upper boundary point (at the $TC_{1.L}$ position) in the CPS TUC1 range, as illustrated in Figure 7.11.

(3.3) *Attaining the final fault diagnosis scope*

Consequently, at the end of the above two stepwise scope reduction processes being applied, we attain the *final fault diagnosis scope* that ranges from the final lower boundary point at the $TC_{1.K}$ position to the final upper boundary point at the $TC_{1.L}$ position in the CPS TUC1 range (as illustrated in Figure 7.11). In the final fault diagnosis scope, test contract TC_TL_GREEN is verified as the precondition assertion just before the execution of operation TO_TL_RED, and test contract TC_TL_RED is verified as the postcondition assertion just after the execution of operation TO_TL_RED.

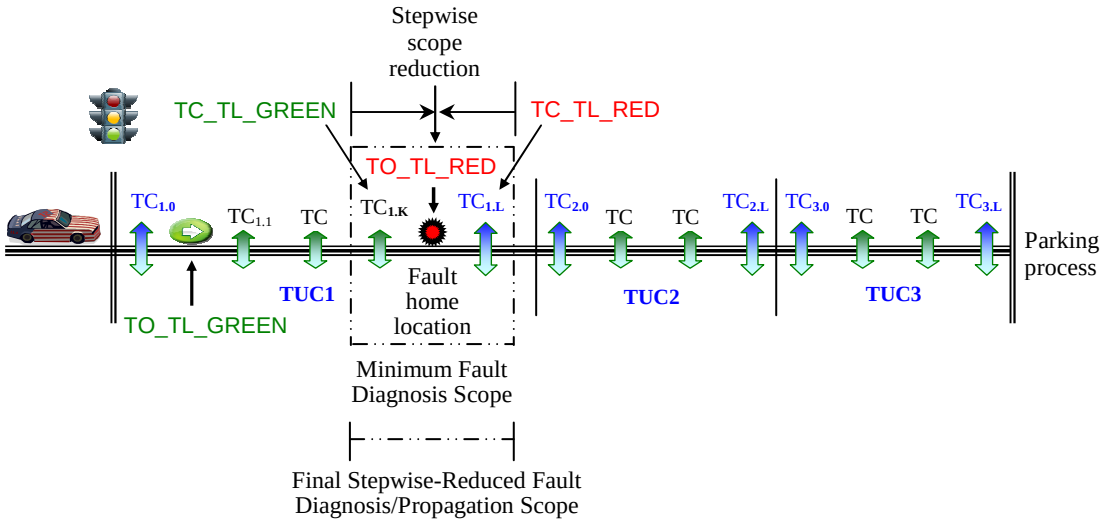


Figure 7.11 CBFDD: Stepwise Fault Diagnosis and Localisation (Step 3.2.3)

(4) Step #4: *Reducing the fault diagnosis scope to class/operation scope*

The above final fault diagnosis scope contains only three test artefacts, including test contract TC_TL_GREEN, test operation TO_TL_RED and test contract TC_TL_RED, all in the CPS TUC1 range (as illustrated in Figure 7.11). The two test contracts are added to the relevant execution path to diagnose and locate the specific target FAULT_TL_RED fault. This implies

that the above final fault diagnosis scope is actually constrained and reduced to be only related to the scope of operation TO_TL_RED of the traffic light device, and thus, in this case, becomes the *final minimum fault diagnosis scope*.

(5) Step #5: *Locating the target fault that has been detected during testing*

With the above final minimum fault diagnosis scope that only contains the above three test artefacts, we can ascertain that the specific target FAULT_TL_RED fault is located in the execution point of operation TO_TL_RED, as illustrated by the following points:

(a) The two added test contracts are specially-designed test artefacts that work as the upper and lower boundary points, and contribute to the above two-sided stepwise scope boundary reduction to identify the *target execution point* that is related to the fault under diagnosis and to produce the final minimum fault diagnosis scope.

(b) With respect to relevant diagnostic functions, test contract TC_TL_RED is the postcondition assertion and is verified just after the execution of operation TO_TL_RED. This test contract examines the relevant test result and evaluates whether this operation is executed correctly. In addition, test operation TO_TL_GREEN acts as the special precondition assertion and is verified before the execution of operation TO_TL_RED. This test contract examines and shows that the abovementioned CPS operational status is maintained consistently for the purpose of rigorous fault diagnosis and localisation. Both test contracts joint support rigorous diagnosis and localisation of the specific target FAULT_TL_RED fault.

(c) In this case, the only target execution point is at the execution point of operation TO_TL_RED, which is the *fault home location* of the specific target FAULT_TL_RED fault.

(6) Step #6: *Correcting and removing the detected fault* FAULT_TL_RED

As the result of Step #1 to Step #5 above, the specific target FAULT_TL_RED fault has been detected and located in the final minimum fault diagnosis scope. We can now conduct fault correction and removal in the following four possible fault cases, which follow Step #6 in the CBFDD guidelines as described in Section 7.5.5 and are equivalent to the possible CPS fault cases as described in Section 7.6.2.2.1.

(a) If the target operation TO_TL_RED is present, but the current CPS execution scenario/path does not actually execute this operation as expected, we need to modify the relevant CPS execution scenario to put this operation in its expected execution path, so that this operation is executed at its correct execution point in the CPS execution path.

(b) If the target operation TO_TL_RED is present and is executed at its correct execution point, but this operation execution is incorrect or fails, for example, due to the incorrect invocation/usage of this operation (e.g. incorrect operation name use, incorrect operation parameters passing). In this case, we need to use the correct invocation/usage of this operation at its correct execution point.

(c) If the target operation TO_TL_RED is present and is executed at its correct execution point, but this operation execution is incorrect or fails, for example, due to the incorrect definition/implementation of this operation in its home class (which consequently causes the incorrect operation execution return/result). In this case, we need to modify and correct the definition and/or implementation of this operation in its home class.

(d) If the target operation TO_TL_RED is not present or the current CPS execution scenario/path does not actually contain this operation, we need to define this operation in the correct class and/or include this operation at its correct execution point.

Note that, for the purpose of effective fault correction and removal, access to more information about component requirements and design specifications for the CPS system may be needed, especially when there is a need to correct the definition and/or implementation of this operation, as described in (c) and (d) above.

7.6.2.3.3 Stepwise Fault Diagnostic Solution

Section 7.6.2.2 (including sub Sections 7.6.2.2.1 and 7.6.2.2.2) describes a direct fault diagnostic solution in the system development environment (which is used as the current testing context). Section 7.6.2.3 (including sub Sections 7.6.2.3.1 and 7.6.2.3.2) discusses a stepwise fault diagnostic solution from the user perspective in the system user operational environment (which is used as the current testing context). We can observe that the stepwise fault diagnostic solution attained by applying all steps with the CBFDD guidelines is equivalent to the direct fault diagnostic solution.

With the stepwise fault diagnostic solution, the relevant CBCTD needs to correctly incorporate the above three test artefacts, which naturally form a special test group and jointly detect, diagnose and locate the specific target FAULT_TL_RED fault in the final minimum fault diagnosis scope (as shown in Figure 7.11). In particular, test operation TO_TL_RED functions equivalently to test operation 3.2 TO in the test group 3.2 TG, which is executed at the above target execution point to exercise the associated CPS operation related to the target fault under diagnosis. Test contract TC_TL_RED functions equivalently to test contract 3.2 ITC in the basic

test group 3.2 TG and is the postcondition assertion verified just after the execution of operation TO_TL_RED. Test contract TC_TL_GREEN functions equivalently to test contract 1.2 ITC in the basic test group 1.2 TG and acts as the special precondition assertion verified before the execution of operation TO_TL_RED. The above analysis shows that, being equivalent to the direct fault diagnostic solution, the stepwise fault diagnostic solution developed with the CBFDD guidelines can accomplish the same diagnostic functions and tasks, and achieve the same CfD goal, in terms of the low-overhead test contract coverage/usage and desired testing effectiveness and efficiency.

The principle of the extended fault causality chain (as described in Section 7.2) indicates that effective component test design must be able to activate a component fault to cause some observable manifestation of failure in order to diagnose and locate a specific fault. In this sense, the relevant CBCTD based on the direct fault diagnostic solution and the stepwise fault diagnostic solution has been shown to be an *effective component test design*. When this relevant CBCTD is used to test the CPS system, it can activate the specific target FAULT_TL_RED fault in the CPS TUC1 integration context, which then causes the actual CPS safety rule failure scenario as described in Section 7.6.2.1. As described in Section 7.6.2, the CBFDD method is able to attain this relevant CBCTD that can effectively diagnose and locate this specific component fault. Therefore, the relevant CBCTD supported with the CBFDD method is an effective component test design for realising the CfD goal.

7.7 Selection of Test Contracts and Testing Points

This section discusses some important open issues about how to effectively apply test contracts to SCT activities with the TbC technique, such as selection, positioning and verification of test contracts and testing points. We introduce and define a set of new useful notions (including the notion of a testing point, a valid testing point, a valid testing range and a consistent valid testing range), and explore their inter-relationships. Some of these notions have been referred to before (especially in Sections 7.5.3, 7.5.5, 7.6.2.2.2, 7.6.2.3.1 and 7.6.2.3.2) and all these notions are now formally defined here with additional discussions.

7.7.1 Selection of Test Contracts

Selection of test contracts is an important open issue for applying test contracts to SCT activities. The importance of this issue is closely related to testing effectiveness and efficiency with the TbC technique. An essential aspect of test contract selection is that the selected test contract must perform its testing functions for a specific testing requirement or a target testing objective, for example:

(a) A test contract is selected to verify a particular software function;

(b) A test contract is selected to detect and diagnose a specific target fault.

A typical approach is to select test contracts from relevant assertion-based preconditions, postconditions and invariants that describe certain contractual relationships for the related component artefact under test, which are all preferred test candidates for test contract selection. On the other hand, the selected test contract should be relatively simple and easy to design and construct, while it also must perform its testing functions correctly. In practice, the tester may have to make some compromises in the selection of test contracts.

Let us look at a testing example with the CPS case study. To find the specific target FAULT_TL_RED fault in the CPS system (as described in Section 7.6.2), the direct fault diagnostic solution uses test contract 1.2 ITC and test contract 3.2 ITC, and equivalently, the stepwise fault diagnostic solution employs test contract TC_TL_GREEN and test contract TC_TL_RED. For the fault diagnostic purpose, test contract 3.2 ITC (or equivalently, TC_TL_RED) must be selected and applied just after the execution of the associated CPS operation (i.e. test operation 3.2 TO `setRed`() or equivalently, TO_TL_RED) that is related to the target fault under diagnosis. This test contract by its nature is verified as the direct, mandatory postcondition assertion to evaluate the relevant test result of this operation execution. Therefore, the selection of this test contract is regarded as the best selection.

On the other hand, test contract 1.2 ITC by its nature (or equivalently TC_TL_GREEN) is actually the direct, mandatory postcondition assertion for test operation 1.2 TO `setGreen`() (or equivalently, TO_TL_GREEN), and should be positioned and verified just after this test operation. However with the direct fault diagnostic solution, test contract 1.2 ITC acts as an additional, indirect precondition assertion, and it does not need to be positioned just before the related test operation 3.2 TO (as described in Section 7.6.2.2.2), due to the support of the basic CPS control state consistency feature (as described in Section 7.6.2.2.1). With the stepwise fault diagnostic solution, test contract TC_TL_GREEN acts as the special precondition assertion positioned before the execution of operation TO_TL_RED to demonstrate that the abovementioned CPS operational status is maintained consistently for the traffic light device (as described in Section 7.6.2.3.2 Step 3.2 (3.2.3) and Step #5). This test contract is also used as the lower boundary point for stepwise scope reduction. The above analysis indicates that the selection of this test contract is an acceptable selection for diagnosing and locating the specific target FAULT_TL_RED fault, in terms of simple test contract design, practical testing effectiveness and efficiency (as described in Section 7.6.2.3.3).

In the case where we use the idea of the TbC test contract criteria based FDD approach for the CfD goal (as described in Section 7.5.3), we can examine the entire CPS TUC1 test sce-

nario (as illustrated earlier in Figure 5.4) or its corresponding overall test sequence (as illustrated earlier in Figure 6.4) for selecting a better test contract. We can observe that test contract 2.5 ETC (or 3.1 ETC) is better selected and positioned at the testing point just before the execution point of test operation 3.2 TO `setRed`(), and has some improved features over test contract 1.2 ITC used in the direct fault diagnostic solution (or equivalently test contract TC_TL_GREEN used in the stepwise fault diagnostic solution). One distinct feature of test contract 2.5 ETC (or 3.1 ETC) is that it can effectively ensure that test operation 3.2 TO is executed in the correct execution context, especially at the correct execution point in the execution path conforming to the overall CPS TUC1 test sequence for achieving the particular testing objective. In particular, test operation 3.2 TO should be executed at its correct execution point just after the current car has successfully finished entering the PAL entry point. This operation execution point is controlled by test contract 2.5 ETC (or 3.1 ETC): when and only when this ETC returns *true*, the current car has successfully passed through the PAL entry point controlled by the in-PhotoCell sensor device; and then test operation 3.2 TO can be executed at its correct execution point to immediately set the traffic light device to the control state of "TL_RED", in order to prevent the next car from entering the PAL. Therefore, the above analysis shows that the selection of test contract 2.5 ETC (or 3.1 ETC) is a better test contract selection over test contract 1.2 ITC (or equivalently TC_TL_GREEN), which can support fault diagnosis and localisation in a more adequate manner. Note that test contract 2.5 ETC (or 3.1 ETC) by its nature has no effect on checking the control state of the traffic light device.

7.7.2 Selection of Testing Points and Valid Testing Range

While the proper selection of test contracts is very important, it is still not adequate for the goal of applying test contracts to SCT activities effectively. Another important issue is where is the right point in the CUT software where the selected test contract should be positioned and verified, in order to make it possible to achieve the desired testing effectiveness and efficiency.

Conceptually, a *testing point* refers to a software point in the CUT software where a relevant software test (e.g. a test contract) may be positioned and verified for software testing. For selection of testing points, a *valid testing point* of a test contract refers to a testing point where the test contract can make a valid testing effect as expected, for example, for the particular testing function or the specific target testing objective. Certainly, test contracts should be applied and verified at the selected valid testing points. The selected test contract would not have a valid testing effect if it were placed at an incorrectly-selected testing point or an *invalid testing point*. For example, to diagnose and locate the specific target FAULT_TL_RED fault as described in Section 7.6.2, test contract TC_TL_GREEN can be positioned at its valid testing point selected just *after* test operation TO_TL_GREEN. However, this test contract has no effect on this test

operation if this test contract is positioned at a (invalid) testing point selected *before* this test operation. Similarly, this test contract has also no effect on test operation TO_TL_RED if this test contract is positioned at a (invalid) testing point selected *after* this test operation.

One common approach is to select valid testing points from possible software locations where certain relevant assertion-based preconditions, postconditions or invariants should hold for the component artefact under test. A *valid testing range* of a test contract refers to a particular testing range between two selected valid testing points, where the test contract can have a valid testing effect at any intermediate valid test point in this testing range. A *consistent valid testing range* of a test contract refers to a particular valid testing range, where the test contract can make an equivalent valid testing effect at any intermediate valid test point in this valid testing range, for example, the test contract should fulfil the equivalent testing requirement or target objective (e.g. obtaining an equivalent state or testing results). This indicates that any other software artefacts or tests in the consistent valid testing range should have no effect on the related test contract. For example, as described in Section 7.6.2.3.2 Step 3.2 (3.2.3), for test contract TC_TL_GREEN, the valid testing range is after the execution point of operation TO_TL_GREEN and before the execution point of operation TO_TL_RED in the CPS TUC1 test scenario. In fact, this valid testing range for test contract TC_TL_GREEN is also a consistent valid testing range, due to the support of the basic CPS control state consistency feature (as described in Section 7.6.2.2.1).

With respect to the important concept of effectual contract scope defined in the TbC technique (as described earlier in Section 6.3.3), we can explore certain inter-relationships among the three important notions (*effectual contract scope*, *valid testing range* and *consistent valid testing range*) as follows:

(a) Conceptually, a consistent valid testing range of a test contract is a valid testing range, but certainly not vice versa.

(b) In principle, the effectual contract scope of a test contract forms a valid testing range, and possibly vice versa, but they are not exactly the same all the time. It is very possible that the entire effectual contract scope of a test contract may comprise several valid testing ranges of the same test contract. In other words, a valid testing range may be just part of the entire effectual contract scope of a particular test contract.

(c) However, the same property described in (b) above may not always apply to the relationship between the effectual contract scope and a consistent valid testing range of a test contract. In particular, a consistent valid testing range of a test contract forms part of the effectual contract scope, but not usually vice versa. In other words, the effectual contract

scope may contain a valid testing range that is not a consistent valid testing range for the same test contract.

(d) The entire effectual contract scope of a test contract comprises the set union of all valid testing ranges of the same test contract.

For the purpose of effective component testing and fault diagnosis, test contracts should only be applied and verified only at the valid testing points selected in the relevant valid test range. It is advantageous to make use of the feature of a consistent valid test range to optimise testing activities and improve testing effectiveness. Accordingly, to effectively apply test contracts to SCT activities with the TbC technique, one important testing task is to analyse and identify all relevant valid testing points, valid testing ranges and/or consistent valid testing ranges.

7.8 Summary and Discussion

This chapter has applied the TbC technique to undertake component fault detection, diagnosis and localisation, which covers the crucial Step TbC4 in the advanced phase of the stepwise TbC working process (as illustrated earlier in Figure 6.1 in Section 6.2). We developed the extended fault causality chain to guide FDD activities and effective component test design. We introduced the important CfD notion, and developed the CBFDD method that further extends the TbC technique to realise the CfD goal. The CBFDD method comprises the two major technical components, the CBFDD process and the CBFDD guidelines, which are further supported with the three fault diagnosis properties, the two stepwise upper/lower-boundary scope reduction strategies and the two associated processes. The CBFDD process establishes the primary foundation of the CBFDD method, and aims to detect and diagnose as many component faults as possible. The CBFDD guidelines provide the series of diagnostic steps, and aim to detect, diagnose and locate target component faults. Then we showed how to apply the complete CBFDD method to fault detection, diagnosis and localisation with the CPS case study. We developed and illustrated the two types of useful and equivalent fault diagnostic solutions (i.e. the direct fault diagnostic solution and the stepwise fault diagnostic solution) with the CBFDD method in the two major possible testing contexts (i.e. the system development environment and the system user operational environment). The illustrative examples have demonstrated that the CBFDD method is capable of supporting effective component test design, diagnosing and locating component faults, and achieving the CfD goal. These are the major contributions of the CBFDD method together with the TbC technique.

There are two main types of FDD approaches with the TbC technique: the CBFDD method and the TbC test contract criteria based FDD approach. The TbC test contract criteria based FDD approach supports high-level coverage of adequate test contracts, and can be applied particularly in conjunction with the overall CBFDD process to systematically detect and diagnose as many new potential component faults as possible. The CBFDD method (especially the CBFDD guidelines) aims to overcome some of the deficiencies of the TbC test contract criteria based FDD approach, and achieve low-overhead test contract coverage and acceptable testing effectiveness and efficiency, which are the main advantages of the CBFDD method.

Therefore, this chapter has shown the FDD-based feature of the MBSCT methodology, which ensures that component test evaluation is achievable in the third phase of the MBSCT framework. At the same time, this chapter has demonstrated and validated the MBSCT testing applicability and capabilities particularly for component fault detection, diagnosis and localisation, and adequate component fault coverage and diagnostic solutions (which are the core MBSCT testing capabilities #3 and #6 as described earlier in Section 4.6). A more comprehensive validation and evaluation of the MBSCT methodology will be presented in Chapter 9.

This chapter has described the FDD-related Step TbC4 in the TbC advanced phase. The TbC technique (especially the CBFDD method) supports effective component test design, making it possible to generate component tests that can attain the desired FDD capability for realising the CfD goal. The subsequent contract-based test generation (i.e. Step TbC5 in the advanced phase of the stepwise TbC working process) will be discussed in Chapter 8.

Chapter 8
Component Test Design and Generation

8.1 Introduction

Component test design and generation in the second phase of the MBSCT framework derives component test cases for UML-based SCT (as described earlier in Section 4.4). The previous chapters of this thesis (Chapter 4 to Chapter 7) have described the methodological foundation and technical aspects of component test development with the MBSCT methodology, including test model construction (in Chapter 5), contract-based component test design (in Chapter 6), and contract-based component fault detection and diagnosis for improving the effectiveness of component test design (in Chapter 7). This chapter mainly focuses on component test generation, which is undertaken with the fifth MBSCT methodological component, the *Component Test Mapping* (CTM) technique. Chapter 4 previously presented a basic introduction to the CTM technique and this chapter goes into much more details about the technical aspects of the CTM technique [167] [169] [171] [172] [173] [174] [175] [176] [179].

In this chapter, Section 8.2 describes the main tasks and techniques for designing and generating component test cases, and reiterates the process and summarises the main technical aspects of component test development with the related MBSCT methodological components before we move on to component test generation. Then, Section 8.3 discusses the CTM technique, including relevant mapping concepts, principles, process and steps as well as mapping criteria. Section 8.4 describes the derivation of the target CTS test case specifications. Section 8.5 gives a summary of this chapter.

8.2 Main Tasks and Techniques

Component test design and generation refers to a process of component test development for SCT, and the major target tasks include:

(a) Analysing and identifying what software artefacts are needed to be tested for the target testing requirements and objectives;

(b) Designing and constructing test sets with test scenarios and test sequences;

(c) Designing and constructing related composite test artefacts in test sequences and test sets;

(d) Conducting component test design for fault detection and diagnosis;

(e) Generating component test cases to evaluate and demonstrate component correctness and quality.

The MBSCT methodology employs a set of useful methodological components to support component test design and generation, and our method of component test development has its own particular technical characteristics to fulfil the above target tasks, as described as follows:

(1) Component test development is *model-based*

Component tests are developed based on relevant UML-based test models constructed in the first phase of the MBSCT framework. In Chapter 5, we discussed how to undertake test model construction with the related MBSCT methodological components, including the integrated SCT process, the TCR strategy, the scenario-based CIT technique and the TbC technique. We also described what test artefacts (including basic test artefacts and special test artefacts) are needed to be identified and designed with test models for UML-based SCT. Test models capture necessary testing-related artefacts and establish the primary foundation for component test design and generation. This feature fulfils the target tasks (a) and (c) above.

(2) Component test development is *scenario-based*

Component tests are developed based on relevant test scenarios that are designed and constructed with the scenario-based CIT technique, in order to test crucial component functional scenarios (e.g. behavioural instances or integration scenarios), as described earlier in Section 4.3.2 and Sections 5.2.2, 5.4.2 and 5.5.2. Test scenarios and associated test sequences are the basis for designing and constructing test sets to organise and group relevant component test cases for a particular testing purpose. This feature fulfils the target task (b) above. Note that more than one test set may be designed in a complex test scenario or test sequence.

(3) Component test development is *contract-based*

Component tests are developed based on relevant test contracts that are the special test artefacts designed and constructed with the TbC technique, in order to bridge the identified "*test gaps*" and improve component testability for effective component test design and generation (as described earlier in Chapter 6). Test contracts are useful testing-support artefacts to enable testing-related component/model artefacts to become testable as required. This feature can effectively enhance component test development with verifiable test artefacts in relevant test sequences and test sets, and further aid in the fulfilment of the target tasks (a) and (c) above.

(4) Component test development is *FDD-based*

Component tests are developed to detect, diagnose and locate component faults for the goal of improving and evaluating component quality. In Chapter 7, we discussed how to apply the TbC technique (especially the CBFDD method) to undertake FDD to enhance the effectiveness of component test design, making it possible to generate component tests that can attain the

desired FDD capability. This feature specifically fulfils the target task (d) above.

(5) Component test development is *process-based*

Component tests are developed based on relevant testing processes that are created and supported with the corresponding MBSCT methodological components. The iterative SCT process provides a process model for the entire MBSCT methodology, and enables the iterative and incremental development of test models and model-based component tests (as described earlier in Chapter 4 and Chapter 5). With the TbC technique (as described earlier in Chapter 6), the stepwise TbC working process shows the main contract-based SCT activities for contract-based component test design and generation. With the CBFDD method (as described earlier in Chapter 7), the CBFDD process guides the main steps for component fault detection and diagnosis, and the two upper/lower-boundary scope reduction processes show the stepwise diagnostic steps to diagnose and locate component faults. With the CTM technique (to be described in the next Section 8.3), the stepwise CTM process shows a series of the steps for test mapping and transformations to generate target component test cases. This feature supports the fulfilment of the target tasks (a) to (e) above.

As discussed above, we can see that our method of component test development is able to achieve the above target tasks for component test design and generation. The related MBSCT methodological components enable component test design to produce adequate test artefacts for effective component test generation. To further the MBSCT methodology, we develop the CTM technique to provide more technical support for component test development, enabling it to become *mapping-based*. This feature particularly fulfils the target task (e) above.

8.3 Component Test Mapping Technique

This section describes the *Component Test Mapping* (CTM) technique, which we develop as the fifth MBSCT methodological component. We introduce the CTM definition, the stepwise CTM process with a series of mapping steps, and the CTM criteria, which are all developed to support the CTM technique to become a new mapping-based approach to component test development. At the same time, we employ the CPS case study to illustrate how to put the CTM technique into practice to undertake component test development, with the focus particularly on component test generation.

8.3.1 The CTM Definition

The CTM technique is developed to be a new mapping-based technique that explores the fundamental relationship between the two domains, SCD artefacts and SCT artefacts with UML

models, in order to support effective component test derivation and to bridge the identified "*test gaps*" between these two domains in UML-based SCT practice. In the context of UML-based SCD, a model artefact may be a scenario, a sequence, a message, an operation, an element, etc. Because UML-based modeling is conducted for component design and implementation, these model artefacts will eventually correspond to, and be realised with, one or more component artefacts in the final component implementation. In the context of UML-based SCT, model-based test artefacts will finally correspond to, and be realised with, one or more testing-related component artefacts that are used for component test derivation. Consequently, following this corresponding testing relationship between these testing-related model/component artefacts in the two domains, developing model-based test artefacts leads to developing corresponding testing-related component artefacts for the SCT purpose. This is one of the crucial principles of model-based testing approaches, which is also the primary basis on which the CTM technique is developed.

The CTM technique establishes a (1 – n) test mapping relationship between the two sets (i.e. the set of model artefacts for UML-based SCD, and the set of testing-related component artefacts for UML-based SCT), which can be described in the following CTM definition (note that the test mapping operation is denoted with the "→" symbol):

Definition 8–1. *Component Test Mapping* is a (1 – n) test mapping relationship between the following two sets:

(1 – n) CTM: SCD_Set → SCT_Set

where SCD_Set = {elements of SCD specifications, e.g. model artefacts for UML-based SCD}; SCT_Set = {elements of SCT specifications, e.g. testing-related component artefacts for UML-based SCT}.

The CTM definition denotes that *one element* in SCD_Set may be mapped, and thus correspond to, *one or more elements* in SCT_Set for deriving and specifying a particular test for a specific testing objective. This indicates that there are two mapping relationships:

(a) (1 – 1) simple mapping relationship (for n = 1): *one element* in SCD_Set is mapped, and thus correspond to, *one element* in SCT_Set;

(b) (1 – n) general mapping relationship (for n > 1): *one element* in SCD_Set is mapped, and thus correspond to, *multiple elements* in SCT_Set.

The CTM technique employs this definition to unify the relevant testing activities in the

complex process of model-based component test development under a single testing concept. An important implication from the CTM definition is that component test derivation needs to focus on how to map and transform relevant model-based test artefacts into useful test case data for deriving target component test cases.

8.3.2 The Stepwise CTM Process

Furthermore, the CTM technique provides a useful systematic mapping process to realise the above CTM definition for practical component test derivation. The CTM process uses a series of mapping steps for test transformations and constructions in terms of different model-based test artefacts towards target component test cases. Figure 8.1 illustrates the stepwise CTM process and the main mapping steps as well as their relationships. Among the six main CTM steps, an earlier "TM" step provides certain test structures and constructs, based on which a later "TM" step derives and complements more specific test data details for generating target component test cases.

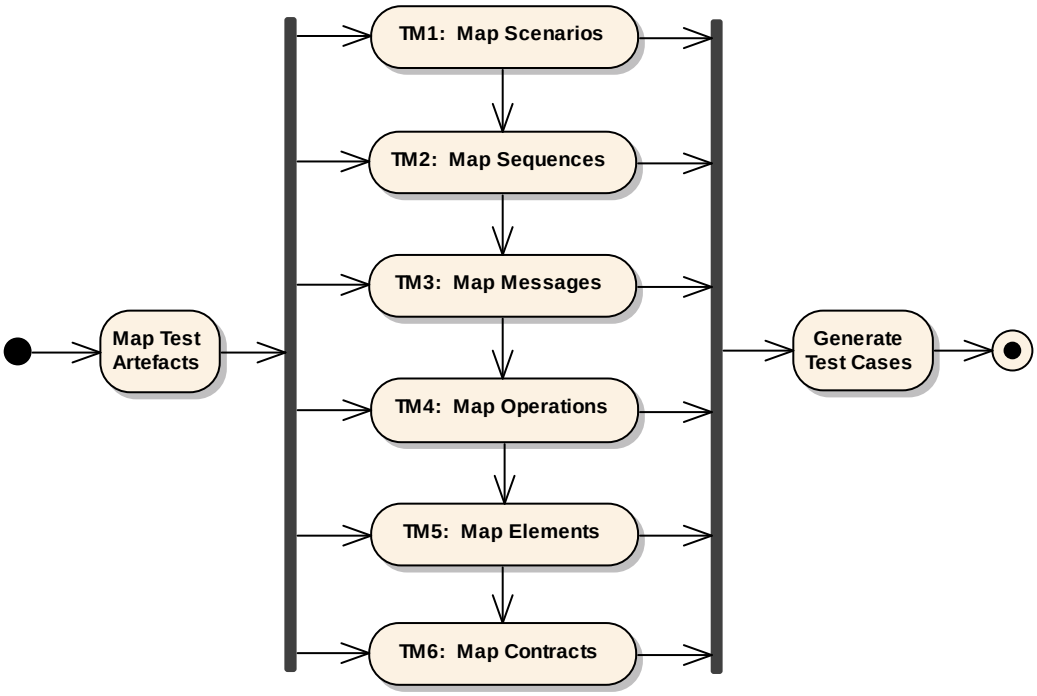


Figure 8.1 The Stepwise CTM Process

Test models are constructed to capture adequate test artefacts to provide the SCT foundation for component test design and generation. The CTM technique refines the process of component test derivation from test models, and provides practical test transformation strategies on how to transform model-based test artefacts to abstract test cases and then to concrete test cases for generating target component test cases. As discussed in Section 1.2, Section 4.3.5 and Ap-

pendix A, the MBSCT methodology employs the XML-based CTS to specify and represent target component test cases. Accordingly, we derive our target component test cases to become CTS test case specifications, which are executable for dynamic testing with the testing-supported software tools developed by the previous SCL project (as described in Appendix A).

The test mapping for deriving the above target component test cases is actually carried out in the two main technical mapping phases, which apply typically in each individual step in the CTM process and are outlined as follows:

(1) CTM Phase #1: The test mapping firstly maps out and produces adequate test artefacts and test data for deriving component test cases.

(2) CTM Phase #2: Then, these test artefacts/data are further mapped to appropriate CTS constructs and elements to generate the target CTS test case specifications.

The first phase described above (i.e. CTM Phase #1) is undertaken mainly with the other MBSCT methodological components in test model construction and model-based component test design (as described earlier in Chapter 4 to Chapter 7). In this section, our test mapping particularly focuses on the second phase (i.e. CTM Phase #2) in each step in the CTM process. Accordingly, the CTM technique undertakes mapping-based component test generation, which maps and transforms relevant UML-based test artefacts and test contracts to abstract test artefacts/data and then generates the target CTS test case specifications for UML-based SCT. The following subsections describe how each step in the CTM process works, and use some examples selected from the CPS case study to illustrate the relevant test mapping details for component test generation. The above two mapping phases can be summarised as shown in Figure 8.2.

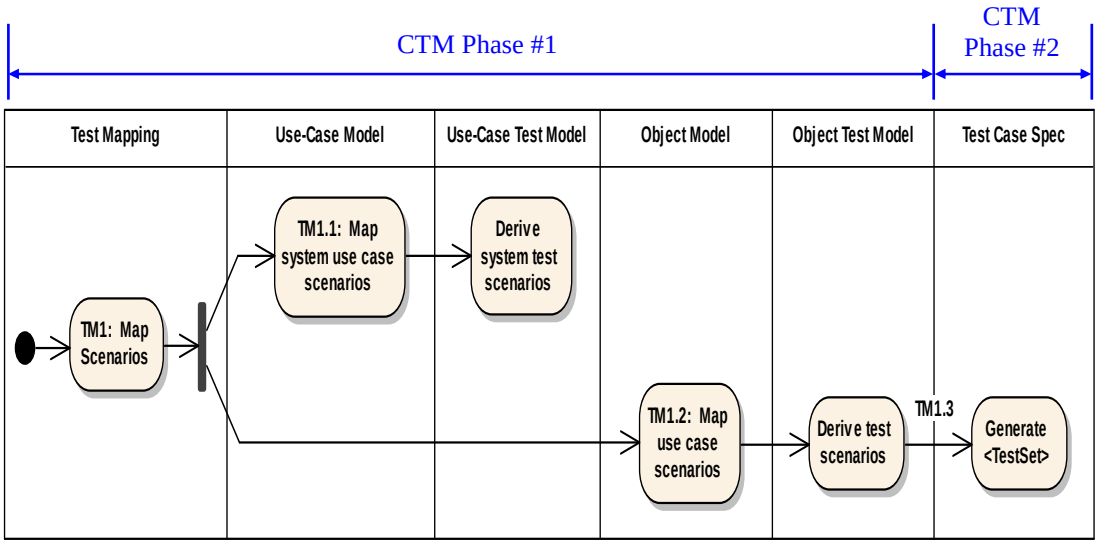
Mapping-Based Component Test Generation		
		{ model-based test artefacts with UML models }
CTM Phase #1:	→	{ abstract test artefacts/data for target test cases }
CTM Phase #2:	→	{ target CTS test case specifications }

Figure 8.2 CTM: Test Mapping Phases

8.3.2.1 TM1: Mapping Scenarios

For component test derivation, *Step TM1* in the CTM process maps and transforms use case scenarios that capture component behaviour with a sequence of interactions and operations for CIT. This scenario mapping takes place at two levels (i.e. the system level and the object level)

in the two CTM phases as shown in Figure 8.3. Note that Figure 8.3 (a) shows Step TM1 in the form of diagrammatic illustration and Figure 8.3 (b) shows Step TM1 in the form of tabular illustration. Both illustrations jointly show the relevant CTM tasks and activities (e.g. sub steps TM1.1, TM1.2, etc.) in Step TM1. Similar illustrations are used for describing each CTM step (Step TM1 to Step TM6).



(a) TM1: Mapping Scenarios (Diagrammatic Illustration)

	TM1:	Mapping Scenarios		
Phase #1:	TM1.1:	system use case scenarios	→	system test scenarios
	TM1.2:	use case scenarios	→	test scenarios
Phase #2:	TM1.3:	test scenarios	→	test sets <TestSet>

(b) TM1: Mapping Scenarios (Tabular Illustration)

Figure 8.3 TM1: Mapping Scenarios

(1) TM1.1: system use case scenarios → system test scenarios

At the system level, a system use case scenario describes how system events/operations interact between the actor and the system, which can be illustrated with system sequence diagrams in the UCM (Use Case Model) (as described earlier in Section 5.4).

The test mapping in Step TM1.1 is a (1 – 1) simple mapping relationship, and mapping a system use case scenario results in a corresponding system test scenario, which is realised and represented with system test events/operations to examine and verify system interactions for system integration testing. The system test scenario can be illustrated with system test sequence diagrams in the UCTM (Use Case Test Model) (as described earlier in Section 5.4). For example, Figure 5.2 in Section 5.4.2 used a system test sequence diagram to illustrate the system test scenario for the CPS TUC1.

(2) TM1.2: use case scenarios → test scenarios

A scenario at the system level is further refined into a scenario at a SCD level subordinate to the system level, such as at the object analysis, design or implementation level. This indicates that the scenario mapping in Step TM1.2 may take place more than once. At the object level, a scenario is a use case instance describing interactions among collaborating objects in the integration context, which can be illustrated with UML sequence diagrams in the object model (as described earlier in Section 5.5).

The test mapping in Step TM1.2 is a (1 – 1) simple mapping relationship, and mapping a scenario produces a corresponding test scenario for undertaking CIT. The test scenario captures a sequence of test messages/operations to examine and verify whether object interactions correctly fulfil the required functions by integrated objects in the integration context. The test scenario can be illustrated with test sequence diagrams in the object test model (as described earlier in Section 5.5). For example, Figure 5.4 in Section 5.5.2 used a design test sequence diagram to illustrate the test scenario for the CPS TUC1 at the object design level.

(3) TM1.3: test scenarios → test sets <TestSet>

After scenarios are mapped out in CTM Phase #1 as described above, a test scenario is further mapped and transformed to a test set (represented with XML element <TestSet>) for generating the target CTS test case specification. This element represents the top level of CTS test sequences under the root element <TestSpecification> in the CTS test case specification. In CTM Phase #2, a complex test scenario may be mapped to more than one test set (i.e., that is a (1 – n) general mapping relationship), which depends on the size and complexity of the actual test scenario. For example, the CPS TUC1 test scenario comprises *three sub test scenarios* (as shown earlier in Figure 5.4 in Section 5.5.2), which can be mapped to the following *three test sets* (as shown in Figure 8.4):

(a) The first test set mapped for sub test scenario #1 describes a set of tests to examine and verify that the traffic light turns to the expected state of "TL_GREEN" before the test car starts entering the PAL, where the relevant CPS operations are controlled by the device control component.

(b) The second test set mapped for sub test scenario #2 describes a set of tests to examine and verify that the test car correctly proceeds, enters and passes through the PAL entry point, where the relevant CPS operations are controlled by the car control component.

(c) The third test set mapped for sub test scenario #3 describes a set of tests to examine and verify that the traffic light turns to the expected state of "TL_RED" after the test car has entered the PAL, where the relevant CPS operations are controlled by the device control component.

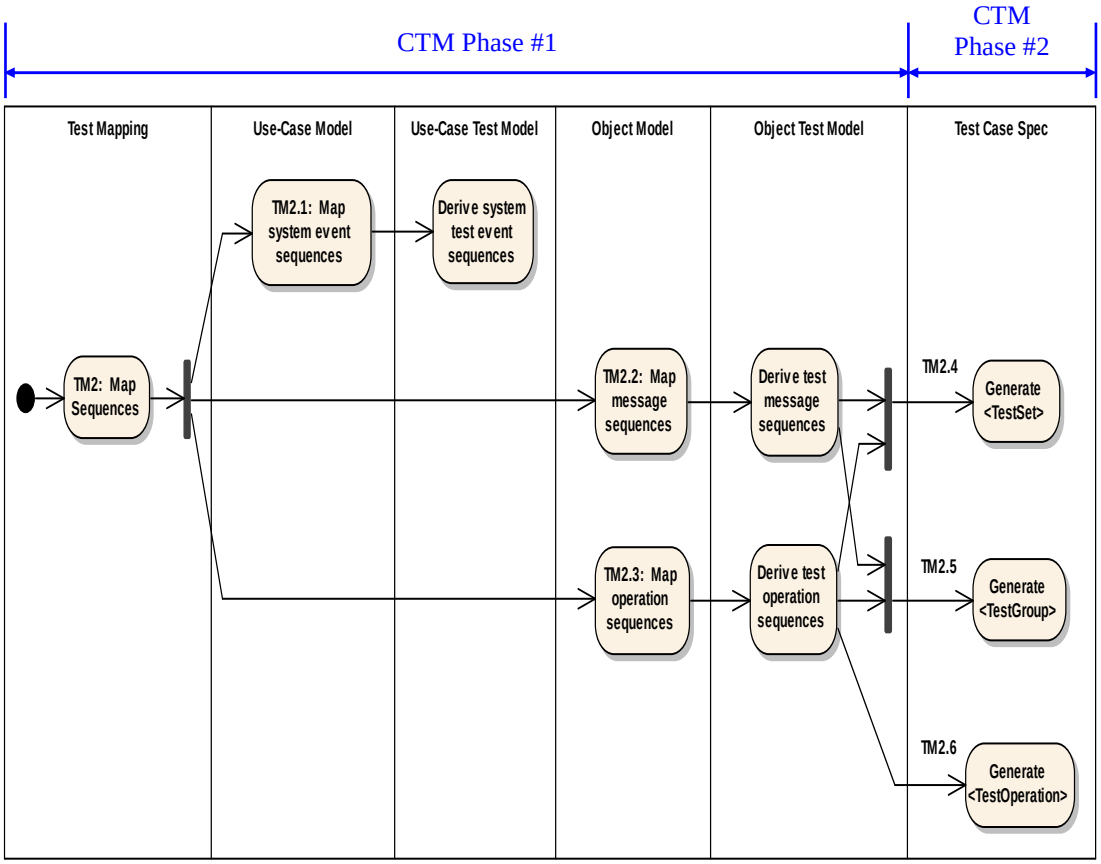
```
... ... ... ...
<TestSpecification Name="CPS_TUC1_CTS.xml">
..<Desc>CTS test case specification for CPS TUC1: car enters PAL</Desc>
... ... ... ...

..<TestSet Name="TUC1_TestSet_turnTLtoGreen">
....<Desc>Test Set #1: this test set examines turning traffic light to the state
          of "TL_GREEN"</Desc>
....<!-- the details of the test set are to be mapped out and constructed -->
..</TestSet>

..<TestSet Name="TUC1_TestSet_carEnterPAL">
....<Desc>Test Set #2: this test set examines car entering the PAL entry point</Desc>
....<!-- the details of the test set are to be mapped out and constructed -->
..</TestSet>

..<TestSet Name="TUC1_TestSet_turnTLtoRed">
....<Desc>Test Set #3: this test set examines turning traffic light to the state
          of "TL_RED"</Desc>
....<!-- the details of the test set are to be mapped out and constructed -->
..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure 8.4 TM1: Overall CTS test sets mapped for the CPS TUC1 test scenario

Note that certain details of test scenarios (e.g. composite test sequences, test messages and/or test operations) are further transformed, constructed and complemented in conjunction with relevant subsequent test mapping steps. Accordingly, certain CTS element details (e.g. XML elements <TestGroup>, <TestOperation>) between the XML elements <TestSet> and </TestSet> for one test set in the CTS test case specification are then produced in conjunction with the relevant subsequent test mapping steps. All these test mapping aspects are further described in the subsequent Sections 8.3.2.2 to 8.3.2.6.

8.3.2.2 TM2: Mapping Sequences

The sequence mapping in *Step TM2* carries out mapping and transforming the sequences of interactions into the sequences of logically ordered composite test artefacts, which are called test sequences. Test sequences realise and represent test scenarios for undertaking CIT, using a sequence of test messages/operations to examine and verify whether object interactions correctly fulfil the required functions by integrated objects in the integration context. A sequence may be composed of logically ordered system events, abstract messages or executable object operations, which all realise interactions occurring at different SCD levels. Accordingly, the sequence mapping may take place to derive test sequences at different mapping levels as shown in Figure 8.5 (a) and Figure 8.5 (b). In particular, Step TM2 results in system test event sequences mapped from system event sequences, test message sequences mapped from message sequences, and test operation sequences mapped from operation sequences.



(a) TM2: Mapping Sequences (Diagrammatic Illustration)

	TM2:	Mapping Sequences		
Phase #1:	TM2.1:	system event sequences	→	system test event sequences
	TM2.2:	message sequences	→	test message sequences
	TM2.3:	operation sequences	→	test operation sequences
Phase #2:	TM2.4:	test sequences	→	test sets <TestSet>
	TM2.5:	test sequences	→	test groups <TestGroup>
	TM2.6:	test sequences	→	test operations <TestOperation>

(b) TM2: Mapping Sequences (Tabular Illustration)

Figure 8.5 TM2: Mapping Sequences

(1) TM2.1: system event sequences → system test event sequences

At the system level, a system event sequence realises and represents a system scenario where the composite system events/operations interact with the system to fulfil certain target system functions, which can be illustrated with system sequence diagrams in the UCM (as described earlier in Section 5.4).

The test mapping in Step TM2.1 is a (1 – 1) simple mapping relationship, and mapping a system event sequence produces a corresponding system test event sequence, which represents a related system test scenario for system integration testing. Similar to system test scenarios, we can use system test sequence diagrams in the UCTM (as described earlier in Section 5.4) to capture a system test event sequence in the related system test scenario. For example, Figure 5.2 in Section 5.4.2 employed a system test sequence diagram to illustrate a system test event sequence in the system test scenario for the CPS TUC1. After the sequence is mapped out, we can describe this system test sequence with a sequence of system test events (initially with abstract textual descriptions) as shown in Figure 8.6. Note that Step TM2.1 works in accordance with the UCTM that treats the entire system under test as a black-box entity at the system testing level (as described earlier in Section 5.4). Accordingly, system test contracts are initially added and applied only at the start and end of this system test sequence, but not within this system test sequence.

```
... ... ... ...
..<Test Contract: stopping bar is in the expected state of "SB_DOWN">
..<test car waits for traffic light to turn to the state of "TL_GREEN">
..<traffic light turns to the state of "TL_GREEN">
..<test car crosses and passes through the PAL entry point>
..<traffic light turns to the state of "TL_RED">
..<Test Contract: traffic light is in the expected state of "TL_RED">
... ... ... ...
```

Figure 8.6 TM2.1: System test event sequences mapped for the CPS TUC1 test scenario

(2) TM2.2: message sequences → test message sequences

A sequence at the system level is further refined into a sequence at a SCD level that is subordinate to the system level, such as at the object analysis, design or implementation level. For example, at the object analysis level, an analysis message sequence comprises interacting messages among collaborating objects in the integration context, and is illustrated with UML sequence diagrams.

The test mapping in Step TM2.2 is a (1 – 1) simple mapping relationship, and mapping a message sequence results in a test message sequence for integration testing. We can use test sequence diagrams in the object test model (as described earlier in Section 5.5) to capture a test message sequence in the related test scenario. For example, for the above system test event sequence as shown in Figure 8.6, after the message sequence is further mapped out, we can use a system test sequence diagram to illustrate *three sub test message sequences* for the CPS TUC1 test scenario. These three sub test sequences realise and represent the three corresponding sub test scenarios, which are mapped out into the three test sets as described in Section 8.3.2.1. After the test mapping in Step TM2.2, we can describe these three sub test sequences with the

three sequences of test messages (initially with abstract textual descriptions) as shown in Figure 8.7.

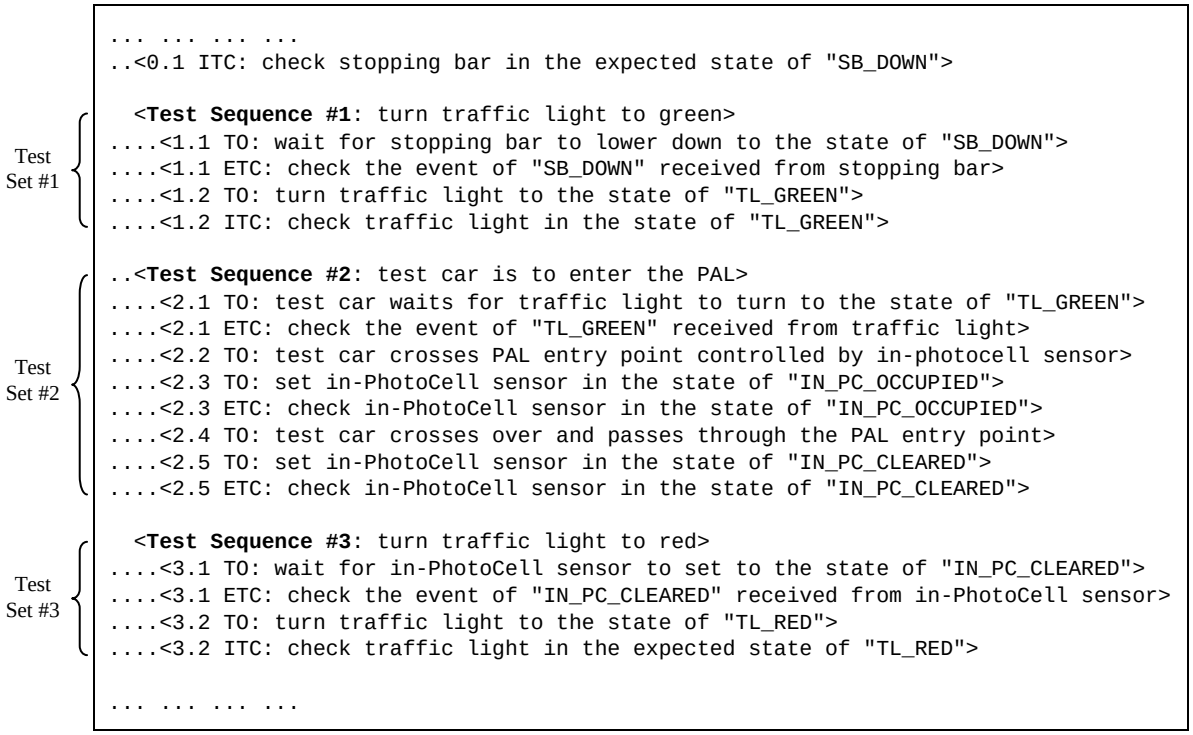


Figure 8.7 TM2.2: test message sequences mapped for the CPS TUC1 test scenario

(3) TM2.3: operation sequences → test operation sequences

At the object design/implementation level, a message is typically represented with one or more operations that fulfil the message. Accordingly, an operation sequence comprises interacting operations among collaborating objects in the integration context, and is illustrated with UML sequence diagrams in the object model (as described earlier in Section 5.5).

The test mapping in Step TM2.3 is a (1 – 1) simple mapping relationship, and mapping an operation sequence produces a test operation sequence for integration testing. We can use test sequence diagrams in the object test model (as described earlier in Section 5.5) to capture a test operation sequence in the related test scenario. For example, Figure 5.4 in Section 5.5.2 used a design test sequence diagram to illustrate *three sub test operation sequences* for the CPS TUC1 test scenario. These three sub test sequences correspond to the three sub test scenarios, which are mapped out into the three test sets as described in Section 8.3.2.1. After the test mapping in Step TM2.3, we can describe these three sub test sequences with the three sequences of concrete test operations and associated test contracts as shown in Figure 8.8. We can observe that a major difference between Step TM2.2 and Step TM2.3 is that the relevant test sequence has been further transformed and refined, and can be represented with the sequence of concrete test operations and associated test contracts in Step TM2.3, rather than with the sequence of test

messages (with abstract textual descriptions) in Step TM2.2.

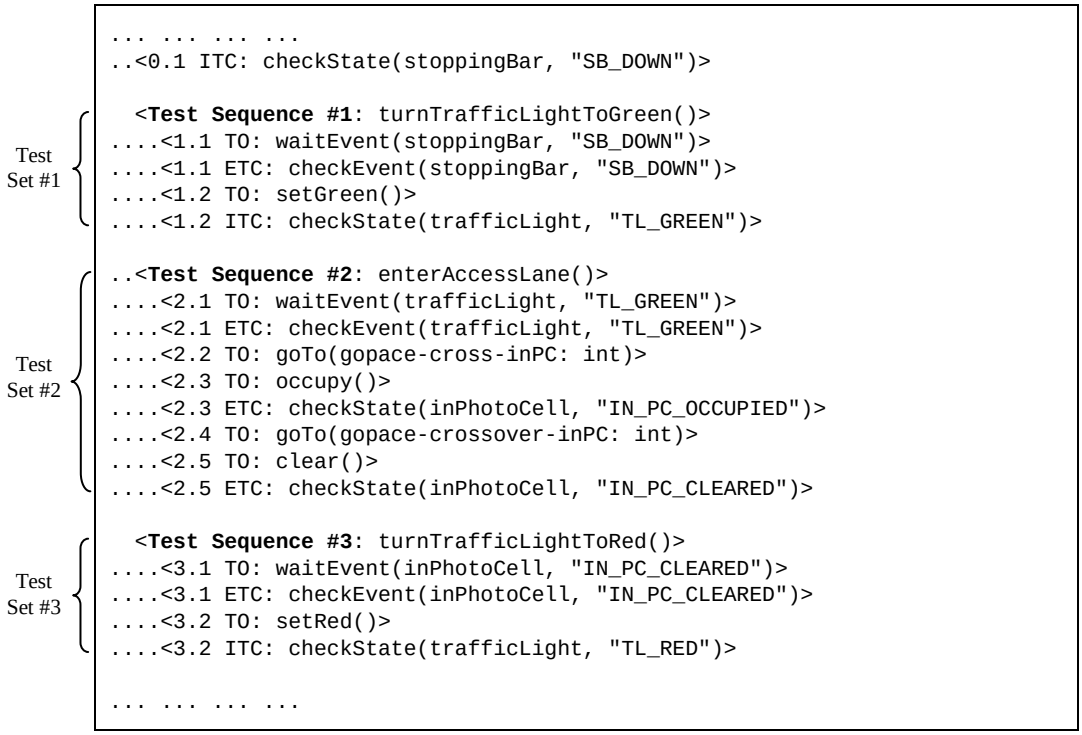


Figure 8.8 TM2.3: test operation sequences mapped for the CPS TUC1 test scenario

In practice, a sequence may cover all messages/operations of the full scenario, or some messages/operations from a partial scenario. Accordingly, a mapped test sequence may be made up of any number of different types of test constituents or units, or the same type of test elements. To deal with these different sequencing situations, our XML-based CTS provides several useful structural elements to construct and represent test sequences at different levels of test granularity to streamline the structure of CTS test case specifications (as described in Section A.2 in Appendix A). After sequences are mapped out in CTM Phase #1 as described above, a test sequence, which is typically composed of multiple test operations and test contracts, needs to be further mapped and transformed to one of the CTS structural elements. Accordingly, this test mapping phase in CTM Phase #2 is required to generate the hierarchical structure of the target CTS test case specification. In the following sub-steps (4) – (6), we show how test sequences are mapped to different types of the CTS structural elements.

(4) TM2.4: test sequences → test sets <TestSet>

As described in Section 8.3.2.1, a test set (represented with XML element <TestSet>), which is typically mapped for a test scenario, represents a sequence of test messages/operations from that test scenario. This CTS structural element may comprise a sequence of subordinate CTS structural elements, such as test groups and/or test operations.

(5) TM2.5: test sequences → test groups <TestGroup>

A test group (represented with XML element <TestGroup>) organises certain related test artefacts together into a special test sequence. As described earlier in Section 6.5, a basic test group is mapped from a pair made up of a test operation and its associated test contract to exercise and verify a particular object interaction in CIT. Several test operations and their associated test contracts may be mapped to one test group if they work closely together for the same testing objective, (e.g. they jointly examine and verify the same complex component interaction). The details of specific test artefacts included in a test group are provided with composite test operations and basic test elements (which are to be further discussed in the subsequent Step TM3 to Step TM6).

(6) TM2.6: test sequences → test operations <TestOperation>

A test operation (represented with XML element <TestOperation>) is the lowest level of the CTS test sequence. Test operations contain specific basic test elements and are used to construct relevant test sequences of test groups or test sets. The test mapping of test operations relates to mapping messages/operations, which is to be further discussed in the subsequent Step TM3 to Step TM4.

We now illustrate by example how to use the different types of CTS structural elements described above to represent test sequences in CTS test case specifications. Taking the CPS TUC1 test scenario as an illustrative example, after the test mapping in Step TM2, we can map out and group relevant test artefacts into a test group that is included in a test set of the CTS test case specification. Figure 8.9 shows three basic test groups selected from the three test sets of the CTS test case specification for the CPS TUC1 test scenario, as described as follows (note that the details of composite test operations and basic test elements are produced in the subsequent test mapping steps):

(a) A basic test group 1.2 TG in the first test set consists of test operation 1.2 TO and its associated test contract 1.2 ITC, which exercises and examines turning the traffic light to the state of “TL_GREEN”.

(b) A test group 2.3 TG in the second test set consists of test operation 2.2 TO, test operation 2.3 TO and its associated test contract 2.3 ETC, which exercises and examines setting the in-PhotoCell sensor device to the state of “IN_PC_OCCUPIED” (i.e. this device senses that the PAL entry point is occupied by the test car).

(c) A test group 3.2 TG in the third test set consists of test operation 3.2 TO and its associated test contract 3.2 ITC, which exercises and examines turning the traffic light to the state of “TL_RED”.

```
... ... ... ...

..<TestSet Name="TUC1_TestSet_turnTLtoGreen">
....<Desc>Test Set #1: this test set examines turning traffic light to the state
          of "TL_GREEN"</Desc>
... ... ... ...

....<TestGroup Name="setGreen_groupedtests">
......<Desc>1.2 TG: grouped tests examine turning traffic light to the state
            of "TL_GREEN"</Desc>
......<TestOperation Name="setGreen_tests">
........<Desc>1.2 TO: examine turning traffic light to the state of "TL_GREEN"</Desc>
........<TestMethod Name="setGreen" ... ...>
..........<Desc>1.2 TO: turn traffic light to the state of "TL_GREEN"</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
........<TestMethod Name="checkState" ... ...>
..........<Desc>1.2 ITC: check traffic light in the resulted correct state
                 of "TL_GREEN"</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
......</TestOperation>
....</TestGroup>

... ... ... ...
..</TestSet>

..<TestSet Name="TUC1_TestSet_carEnterPAL">
....<Desc>Test Set #2: this test set examines car entering the PAL entry point</Desc>
... ... ... ...

....<TestGroup Name="occupy_groupedtests">
......<Desc>2.3 TG: grouped tests examine setting in-PhotoCell sensor in
             the state of "IN_PC_OCCUPIED"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.2 TO: examine the test car crossing the PAL entry point</Desc>
........<TestMethod Name="goTo" ... ...>
..........<Desc>2.2 TO: the test car crosses the PAL entry point controlled by
                 in-PhotoCell sensor</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
......</TestOperation>
......<TestOperation Name="occupy_tests">
........<Desc>2.3 TO: examine setting in-PhotoCell sensor to the state
               of "IN_PC_OCCUPIED"</Desc>
........<TestMethod Name="occupy" ... ...>
..........<Desc>2.3 TO: set in-PhotoCell sensor in the state
                 of "IN_PC_OCCUPIED"</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
........<TestMethod Name="checkState" ... ...>
..........<Desc>2.3 ETC: check in-PhotoCell sensor in the resulted correct
                 state of "IN_PC_OCCUPIED"</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
......</TestOperation>
....</TestGroup>

... ... ... ...
..</TestSet>

..<TestSet Name="TUC1_TestSet_turnTLtoRed">
....<Desc>Test Set #3: this test set examines turning traffic light to the state
          of "TL_RED"</Desc>
```

```
....<TestGroup Name="setRed_groupedtests">
......<Desc>3.2 TG: grouped tests examine turning traffic light to the state
            of "TL_RED"</Desc>
......<TestOperation Name="setRed_tests">
........<Desc>3.2 TO: examine turning traffic light to the state of "TL_RED"</Desc>
........<TestMethod Name="setRed" ... ...>
..........<Desc>3.2 TO: turn traffic light to the state of "TL_RED"</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
........<TestMethod Name="checkState" ... ...>
..........<Desc>3.2 ITC: check traffic light in the resulted correct state
                 of "TL_RED"</Desc>
..........<!-- the details of the test operation/method are to be mapped out
                and constructed -->
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>
... ... ... ...
```

Figure 8.9 TM2: CTS test sequences (test sets/groups/operations) mapped for the CPS TUC1 test scenario

8.3.2.3 TM3: Mapping Messages

Step TM3 maps and transforms interacting messages into test messages to exercise and verify interactions for CIT. Messages may occur in the form of system events, abstract messages, or object operations, and the last form of messages is more useful for realising executable messages. Accordingly, the message mapping may take place to derive test messages at different mapping levels as shown in Figure 8.10 (a) and Figure 8.10 (b). In particular, Step TM3 results in system test messages mapped from system interaction messages, component test messages mapped from component interaction messages, and object test messages mapped from object interaction messages.

(1) TM3.1: system interaction messages → system test messages

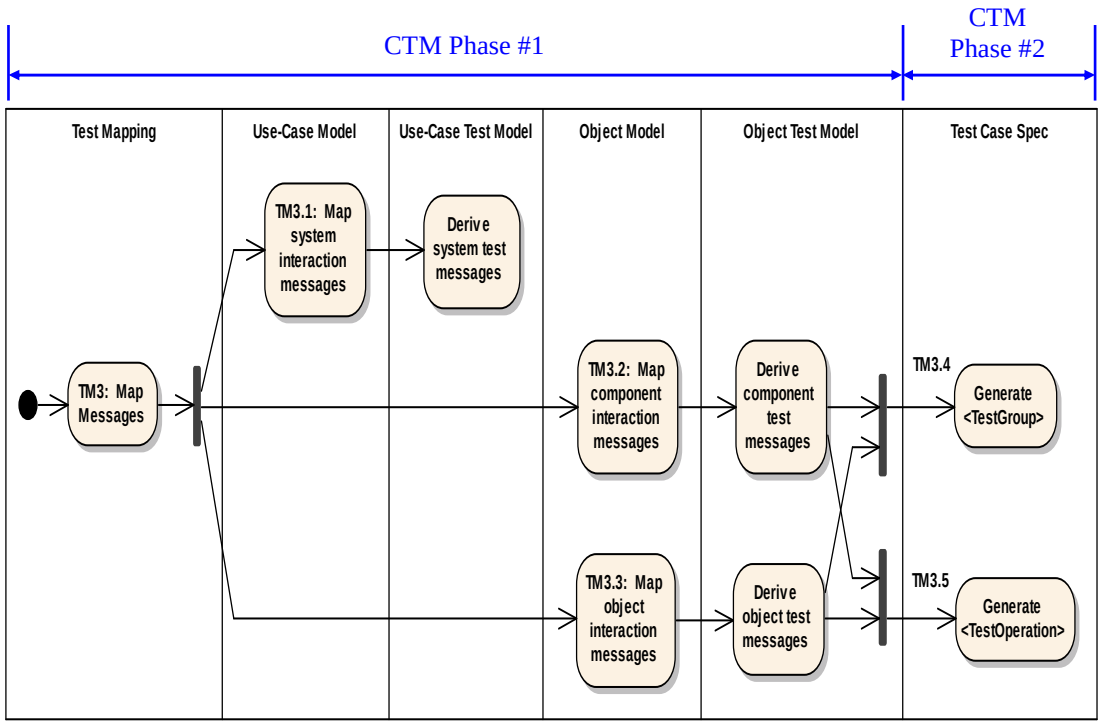
At the system level, a system interaction message represents and fulfils a system interaction as part of a system scenario/sequence. Mapping a system interaction message produces one or more corresponding system test messages as part of the mapped system test scenario /sequence.

(2) TM3.2: component interaction messages → component test messages

At the component level, a component interaction message represents and fulfils a component interaction as part of a component scenario/sequence. Mapping a component interaction message produces one or more corresponding component test messages as part of the mapped component test scenario/sequence.

(3) TM3.3: object interaction messages → object test messages

At the object level, an object interaction message represents and fulfils an object interaction as part of an object scenario/sequence. Mapping an object interaction message produces one or more corresponding object test messages as part of the mapped object test scenario/sequence.



(a) TM3: Mapping Messages (Diagrammatic Illustration)

	TM3:	Mapping Messages		
Phase #1:	TM3.1:	system interaction messages	→	system test messages
	TM3.2:	component interaction messages	→	component test messages
	TM3.3:	object interaction messages	→	object test messages
Phase #2:	TM3.4:	test messages	→	test groups <TestGroup>
	TM3.5:	test messages	→	test operations <TestOperation>

(b) TM3: Mapping Messages (Tabular Illustration)

Figure 8.10 TM3: Mapping Messages

With UML modeling, a message is a specification of a communication or interaction between participating objects, which conveys collaboration information with certain expected activity. A message sent from one object A (called the message's sender object) usually invokes the execution of an operation on another object B (called the message's receiver object); and if

applicable, the operation execution on object B may return some value as a response to the operation-invocation request made by object A (also called the operation's caller or invocator). The class operation, which realises its corresponding interaction message, is statically defined in the UML class diagram and is dynamically instantiated in the UML interaction diagram (e.g. UML sequence diagram). The dynamic information associated with the interaction message usually includes the actual values bound to the operation parameters, and if applicable, the actual result returned from the operation execution. Such dynamic information and binding mechanisms are useful for mapping and constructing corresponding test messages to examine and verify how the class operation realises the interaction message and works collaboratively with other messages in the corresponding scenario/sequence. For example, Figure 8.6 shows test messages in the form of system test events for the CPS TUC1 test scenario. Figure 8.7 shows test messages in the form of object test messages to examine component/object messages for the CPS TUC1 test scenario. Figure 8.8 shows test messages in the form of concrete test operations and associated test contracts to examine component/object operations for the CPS TUC1 test scenario. These three figures (i.e. Figure 8.6, Figure 8.7 and Figure 8.8) actually show the three main forms of abstract test cases derived from the relevant model-based test artefacts towards the target component test cases for the CPS TUC1 test scenario.

After messages are mapped out in CTM Phase #1 as described above, a test message, depending on its complexity, needs to be further mapped to one or more CTS elements to create the related test sequences, which is undertaken in CTM Phases #2 as described below. Figure 8.9 shows some relevant mapping examples selected from the CPS TUC1 test scenario.

(4) TM3.4: test messages → test groups <TestGroup>

A complex test message is mapped to a test group (that contains a number of test operations), which examines and verifies the test message (e.g. test group 1.2 TG, test group 2.3 TG and test group 3.2 TG as shown in Figure 8.9).

(5) TM3.5: test messages → test operations <TestOperation>

A simple test message is mapped to a test operation to examine and verify the test message (e.g. test operation 1.2 TO, test operation 2.2 TO, test operation 2.3 TO and test operation 3.2 TO as shown in Figure 8.9). Test operations mapped from component/class operations are a crucial focus of component test mapping, which is further discussed in the next subsection.

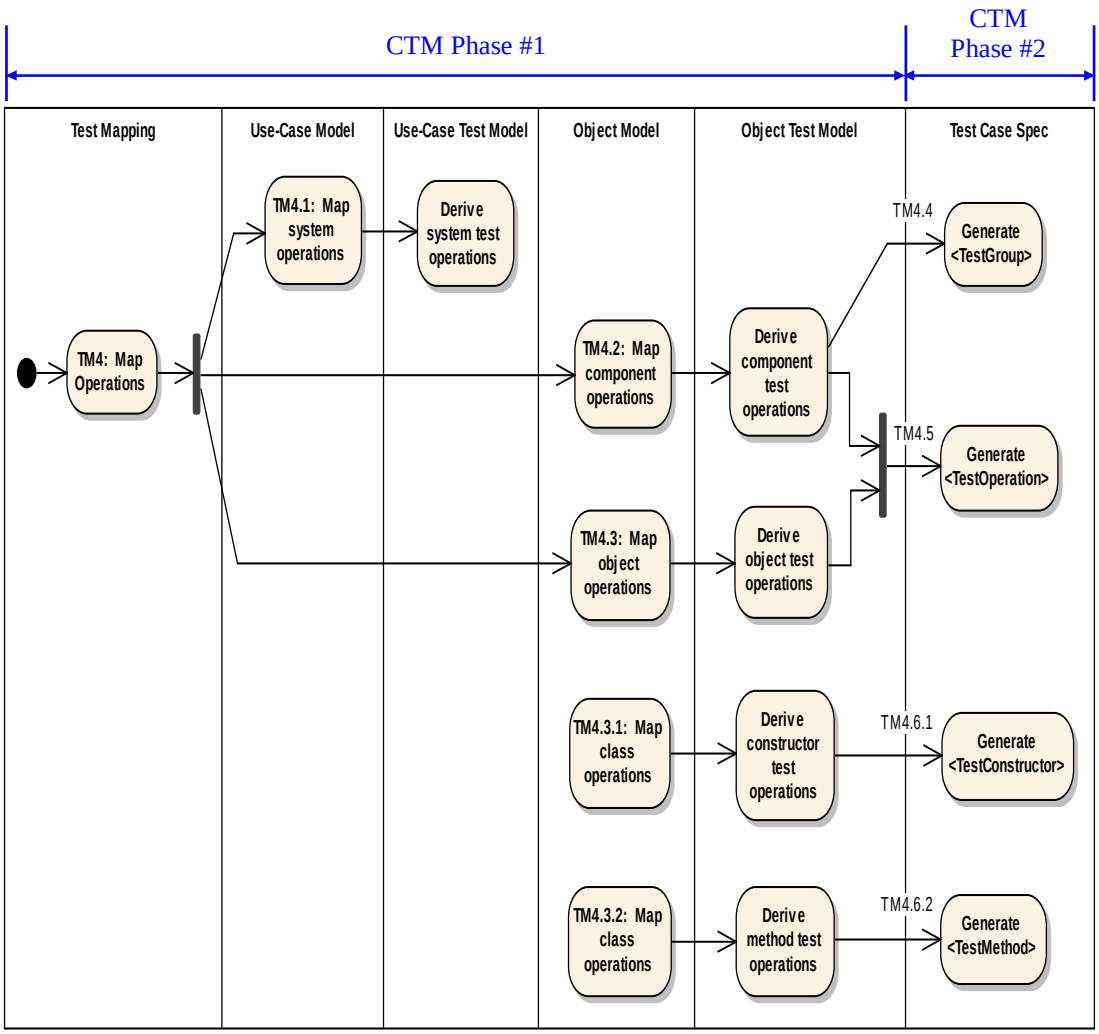
8.3.2.4 TM4: Mapping Operations

Step TM4 is an essential and most useful test mapping step in the CTM process. An operation usually represents and realises a function or behavioural responsibility of a software module

(e.g. a class or component). The operation mapping carries out mapping and transforming functional operations to test operations to exercise and verify whether a particular operation correctly fulfils its target function. Operations may occur at different SCD levels, named as system operations, component operations, object operations, or specific class constructors or methods. Accordingly, the operation mapping may take place to derive test operations at different mapping levels as shown in Figure 8.11 (a) and Figure 8.11 (b). In particular, Step TM4 results in system test operations mapped from system operations, component test operations mapped from component operations, object test operations mapped from object operations, and specific constructor test operations mapped from class constructors or specific method test operations mapped from class methods.

(1) TM4.1: system operations → system test operations

At the system level, a system operation represents and fulfils the full, or a partial, system function. Mapping a system operation may produce one or more corresponding system test operations to verify the related system function.



(a) TM4: Mapping Operations (Diagrammatic Illustration)

	TM4:	Mapping Operations		
Phase #1:	TM4.1:	system operations	→	system test operations
	TM4.2:	component operations	→	component test operations
	TM4.3:	object operations	→	object test operations
	TM4.3.1:	class operations	→	constructor test operations
	TM4.3.2:	class operations	→	method test operations
Phase #2:	TM4.4:	test operations	→	test groups <TestGroup>
	TM4.5:	test operations	→	test operations <TestOperation>
	TM4.6.1:	constructor test operations	→	atomic test operations <TestConstructor>
	TM4.6.2:	method test operations	→	atomic test operations <TestMethod>

(b) TM4: Mapping Operations (Tabular Illustration)

Figure 8.11 TM4: Mapping Operations

(2) TM4.2: component operations → component test operations

At the component level, a component operation represents and fulfils the full, or a partial, component function. Mapping a component operation may produce one or more corresponding component test operations to verify the related component function.

(3) TM4.3: object operations → object test operations

At the object level, an object operation represents and fulfils the full, or a partial, object function. Mapping an object operation may produce one or more corresponding object test operations to verify the related object function.

Usually, an object operation is an instance of the corresponding class operation. Accordingly, at the class level, a class operation (i.e. a class constructor or class method) represents and fulfils the full, or a partial, class function implemented with the class. Mapping a class operation may produce one or more corresponding class test operations to verify the related class function, which is described as follows:

(3.1) TM4.3.1: class constructors → constructor test operations

Mapping a class constructor may produce a single corresponding constructor test operation.

(3.2) TM4.3.2: class methods → method test operations

Mapping a class method may produce one or more corresponding method test operations, depending on the complexity of the class method under test.

In practice, how Step TM4 works to produce each type of test operation depends on the types (e.g. operation level) and complexity of the operations under test. After operations are mapped out in CTM Phase #1 as described above, a test operation needs to be further mapped to one or more CTS atomic test operations and their associated elements to produce the target CTS test case specification. Following the defined CTM relationship, we need to consider the following mapping cases in CTM Phase #2:

(a) (1 – 1) simple mapping relationship

One operation in SCD_Set is mapped and corresponds to one atomic test operation (represented with XML element <TestMethod> or <TestConstructor>) in SCT_Set. In this case, one operation is examined with a test specified with one CTS test operation element. This case often occurs when the operation under test is a simple object operation (e.g. a class method). For example in the CPS TUC1 test scenario, after Step TM4.3.2 and Step TM4.6.2 are carried out, atomic test operation 2.2 TO <TestMethod> (as shown in Figure 8.12) examines class/object operation goTo() for car moving.

(b) (1 – n) general mapping relationship

One operation in SCD_Set is mapped and corresponds to several atomic test operations (represented with XML element <TestMethod> or <TestConstructor>) in SCT_Set. In this case, one operation is examined with several tests specified with several CTS test elements. These generated tests are then structured and organised into certain test sequences made up of related structural elements <TestGroup> or <TestOperation> as necessary. This case may occur when the operation under test is a complex component operation or integration interaction. For example, after Step TM4.2 and Step TM4.4 are carried out, test group 2.3 TG <TestGroup> (as shown in Figure 8.12) is generated and composed of three atomic test operations, which jointly exercise and examine the composite operation that the in-PhotoCell sensor device senses that the PAL entry point is occupied by the test car and is set to the state of "IN_PC_OCCUPIED".

(4) TM4.4: test operations → test groups <TestGroup>

In accordance with the (1 – n) general mapping relationship, a test operation is mapped to a CTS test element <TestGroup>, which may further enclose several basic CTS test elements, such as atomic test operations.

```
... ... ... ...
..<TestSet Name="TUC1_TestSet_carEnterPAL">
....<Desc>Test Set #2: this test set examines car entering the PAL entry point </Desc>
... ... ... ...

....<TestGroup Name="occupy_groupedtests">
......<Desc>2.3 TG: grouped tests examine setting in-PhotoCell sensor in
            the state of "IN_PC_OCCUPIED"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.2 TO: examine the test car crossing the PAL entry point</Desc>
........<TestMethod Name="goTo" Target="testCar">
..........<Desc>2.2 TO: the test car crosses the PAL entry point controlled by
                in-PhotoCell sensor</Desc>
..........<Arg Name="gopace" Source="gopace-cross-inPC" DataType="int" />
........</TestMethod>
......</TestOperation>
......<TestOperation Name="occupy_tests">
........<Desc>2.3 TO: examine setting in-PhotoCell sensor in the state
              of "IN_PC_OCCUPIED"</Desc>
........<TestMethod Name="occupy" Target="inPhotoCell">
..........<Desc>2.3 TO: set in-PhotoCell sensor in the state
                of "IN_PC_OCCUPIED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="inPhotoCell">
..........<Desc>2.3 ETC: check in-PhotoCell sensor in the resulted correct
                state of "IN_PC_OCCUPIED"</Desc>
..........<Arg Name="aObservable" Source="inPhotoCell"
               DataType="java.util.Observable" />
..........<Arg Name="aState" Source="IN_PC_OCCUPIED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

... ... ... ...
..</TestSet>
... ... ... ...
```

Figure 8.12 TM4: CTS test groups, test operations, test contracts and basic test elements mapped for the CPS TUC1 test scenario

(5) TM4.5: test operations → test operations <TestOperation>

In accordance with the (1 – 1) simple mapping relationship, a test operation is mapped to a CTS test element <TestOperation>, which may further enclose one or more basic CTS test elements, such as atomic test operations.

(6.1) TM4.6.1: constructor test operations → atomic test operations <TestConstructor>

In accordance with the (1 – 1) simple mapping relationship, a constructor test operation is mapped to a CTS test element <TestConstructor>.

(6.2) TM4.6.2: method test operations → atomic test operations <TestMethod>

In accordance with the (1 – 1) simple mapping relationship, a method test operation is mapped to a CTS test element <TestMethod>.

8.3.2.5 TM5: Mapping Elements

Elements represent atomic constituents of software artefacts. An element often holds some specific data that may determine the representation and certain behaviour or functions of the software artefact (e.g. the operation under test). Typically, such specific element data may correspond to an operation's identification (operation name), actual parameter values (acting as test inputs), and/or expected return results (acting as expected test outputs), which are the lowest-level test data used for test generation.

Step TM5 is at the lowest level of test mapping in the entire CTM process. This means that all CTM steps would, in one way or another, eventually reach this final test mapping step in order to complete the mapping activity and derive final test data for generating the target component test cases. As any level of software artefacts may comprise elements that will be useful for software testing, the element mapping may take place to derive test elements at different mapping levels as shown in Figure 8.13 (a) and Figure 8.13 (b). Specifically, Step TM5 may produce system test elements mapped from system elements, component test elements mapped from component elements, object test elements mapped from object elements, and operation test elements mapped from operation elements.

(1) TM5.1: system elements → system test elements

At the system level, system elements are basic constituents to compose (part of) system artefacts, such as system scenarios, system sequences, system events/operations, system contracts, etc. Mapping a system element may produce one or more system test elements to examine and verify the related system artefact.

System test elements are basic test constituents to construct (part of) a particular system test artefact, which works in the following ways:

(a) A system test event/operation is composed of one or more mapped test elements.
(b) A system test contract is composed of one or more related special test operations, which are further composed of the mapped test elements.
(c) A system test sequence is composed of one or more composite test operations and associated test contracts, which are further composed of the mapped test elements.
(d) A system test scenario is composed of one or more related test sequences and test operations/contracts, which are further composed of the mapped test elements.

(2) TM5.2: component elements → component test elements

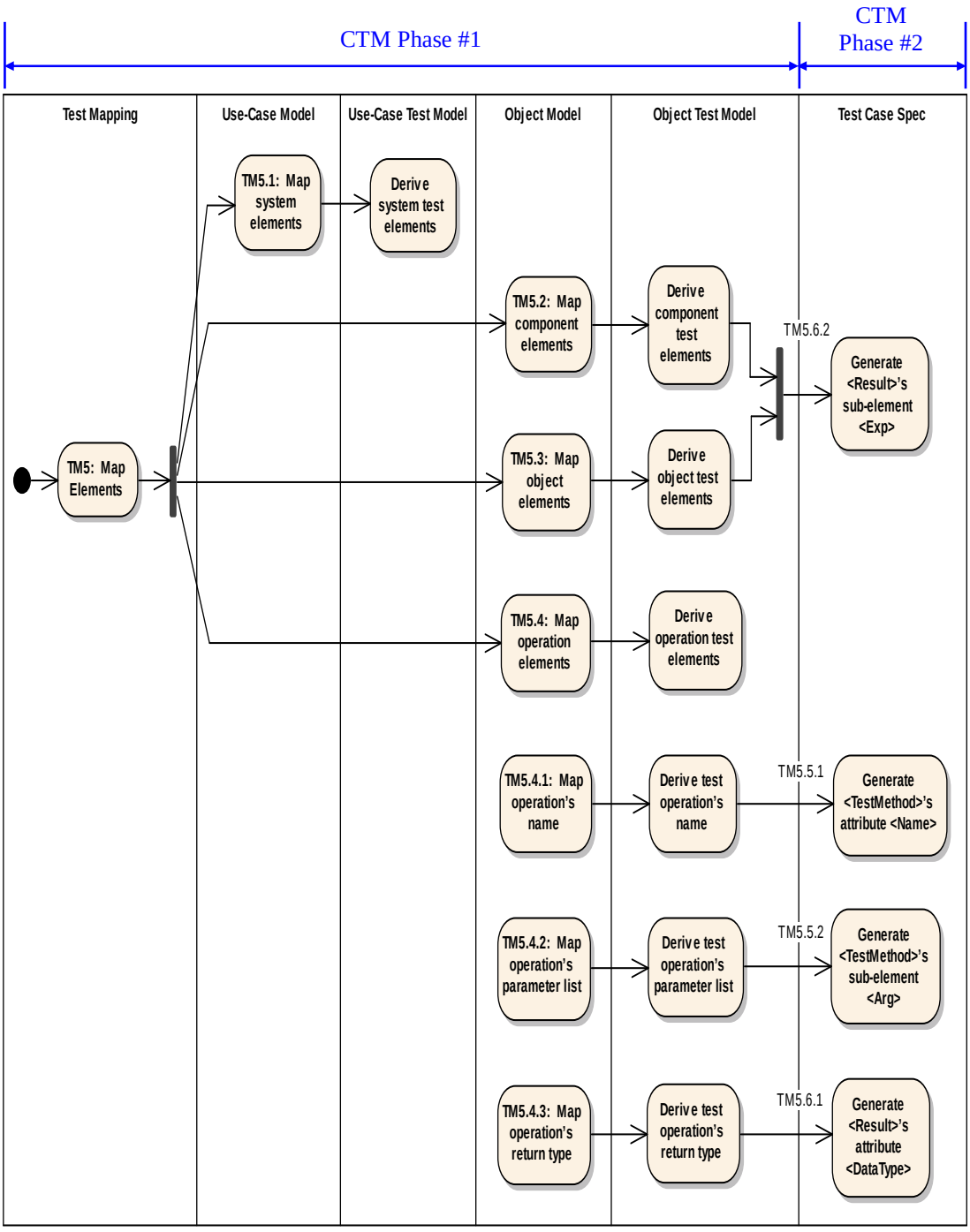
At the component level, component elements are basic constituents to compose (part of) component artefacts, such as component scenarios, component messages/operations, component contracts, etc. Mapping a component element may produce one or more component test elements to examine and verify the related component artefact.

Component test elements are basic test constituents to construct (part of) a particular component test artefact, which works in the following ways:

(a) A component test operation is composed of one or more mapped test elements.

(b) A component test contract is composed of one or more related special test operations, which are further composed of the mapped test elements.

(c) A component test scenario is composed of one or more related test sequences and test operations/contracts, which are further composed of the mapped test elements.



(a) TM5: Mapping Elements (Diagrammatic Illustration)

	TM5:	Mapping Elements		
Phase #1:	TM5.1:	system elements	→	system test elements
	TM5.2:	component elements	→	component test elements
	TM5.3:	object elements	→	object test elements
	TM5.4:	operation elements	→	operation test elements
	TM5.4.1:	operation's name	→	test operation's name
	TM5.4.2:	operation's parameter list	→	test operation's parameter list
	TM5.4.3:	operation's return type	→	test operation's return type
Phase #2:	TM5.5:	test operation elements	→	<TestMethod>'s attributes and sub-elements
	TM5.5.1	test operation's name	→	<TestMethod>'s attribute <Name>
	TM5.5.2	test operation's parameter list	→	<TestMethod>'s sub-element <Arg>
	TM5.6:	test operation elements	→	<Result>'s attributes and sub-elements
	TM5.6.1:	test operation's return type	→	<Result>'s attribute <DataType>
	TM5.6.2:	test operation's expected return result	→	<Result>'s sub-element <Exp>

(b) TM5: Mapping Elements (Tabular Illustration)

Figure 8.13 TM5: Mapping Elements

(3) TM5.3: object elements → object test elements

At the object level, object elements are basic constituents to compose (part of) object artefacts, such as object variables (state/event), object operations, object contracts, etc. Mapping an object element may produce one or more object test elements to examine and verify the related object artefact.

Object test elements are basic test constituents to construct (part of) a particular object test artefact, which works in the following ways:

(a) An object test state/event is composed of one or more mapped test elements.

(b) An object test operation is composed of one or more mapped test elements.

(c) An object test contract is composed of one or more related special test operations, which are further composed of the mapped test elements.

(4) TM5.4: operation elements → operation test elements

At the operation level, operation elements are atomic constituents to compose a specific operation. Mapping an operation element may produce an operation test element to examine and verify the operation under test.

Operation test elements are atomic test constituents to construct basic test data, which are further used to produce the corresponding test operation for verifying the operation under test.

(4.1) TM5.4.1: operation's name → test operation's name

The element for the operation's name is mapped to the test operation's name.

(4.2) TM5.4.2: operation's parameter list → test operation's parameter list

The element for the operation's parameter list is mapped to the test operation's parameter list. The test mapping should maintain the logical, sequential order of parameters in the list, so that each actual parameter value is correctly bound to its corresponding formal parameter for dynamic testing.

(4.3) TM5.4.3: operation's return type → test operation's return type

The element for the operation's return type is mapped to the test operation's return type. An operation may have the explicit return type when it needs to return an actual execution result, or has no return value when its return type is defined as void. Checking return type is part of the verification of the result of the operation's execution.

Operation test elements are at the lowest level of test artefacts that are used as basic test data to construct certain useful test cases. This means that all sub-steps of the element mapping eventually arrive at Step TM5.4 (i.e. mapping operation elements as described in (4) above) so as to complete the mapping activity and derive final test data for generating target component test cases. After operation elements are mapped out in CTM Phase #1 as described above, a test element needs to be further mapped to one or more constituents of a specific XML-based CTS element (especially to the atomic test operation element <TestMethod> and associated sub-elements), in order to generate the target CTS test case specification. This is undertaken in CTM Phases #2 as described below. Figure 8.12 shows some relevant mapping examples selected from the CPS TUC1 test scenario.

(5) TM5.5: test operation elements → <TestMethod>'s attributes and sub-elements

The mapped elements for a test operation are further mapped to the attributes and sub-elements of the atomic test operation element <TestMethod> in the target CTS test case specifi-

cation.

(5.1) TM5.5.1: test operation's name → <TestMethod>'s attribute <Name>

The test operation's name is mapped to the attribute <Name> of the <TestMethod> element.

(5.2) TM5.5.2: test operation's parameter list → <TestMethod>'s sub-element <Arg>

The test operation's parameter list is mapped to the sub-element <Arg> of the <TestMethod> element. Each parameter in the parameter list is mapped to one sub-element <Arg>. All <Arg> sub-elements are mapped and arranged in accordance with the same sequence of the corresponding parameters in the parameter list.

(6) TM5.6: test operation elements → <Result>'s attributes and sub-elements

Verifying operations are required to check the actual execution result and compare it with the expected result. This testing is specified and conducted with the <Result> element's attributes and sub-elements. Note that the <Result> element is a sub-element of CTS elements <TestMethod> and <TestConstructor>.

(6.1) TM5.6.1: test operation's return type → <Result>'s attribute <DataType>

The test operation's return type is mapped to the attribute <DataType> of the <Result> element. When the attribute <Save> of the <Result> element is set to "Y", the actual returned value is recorded for test evaluation.

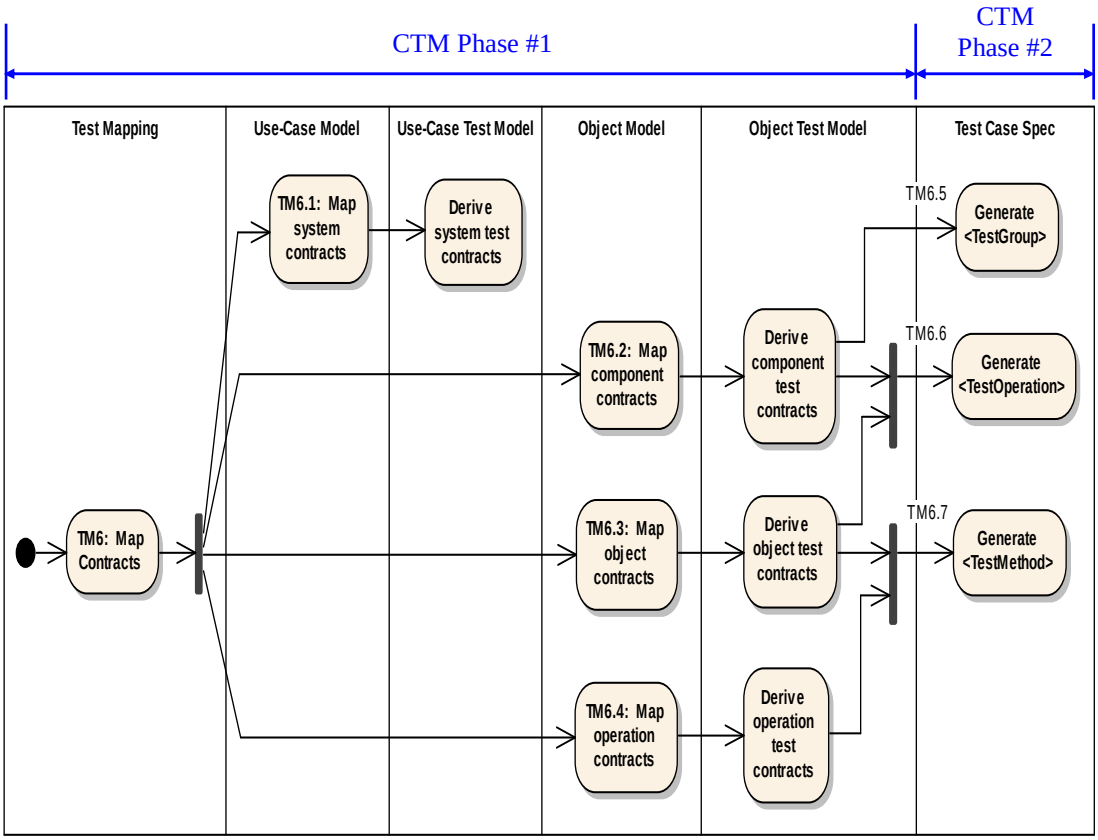
(6.2) TM5.6.2: test operation's expected return result → <Result>'s sub-element <Exp>

The sub-element <Exp> of the <Result> element is used to record and specify the expected return result of the test operation. This mapping is assisted with test contracts that are constructed and applied to the operation under test.

8.3.2.6 TM6: Mapping Contracts

The contract mapping in *Step TM6* maps and transforms contract artefacts to test contracts and then to test operations. This fulfils the final Step TbC5 in the TbC advanced phase of the stepwise TbC working process (as shown earlier in Figure 6.1). With the TbC technique (as described earlier in Chapter 6 and Chapter 7), test contracts are identified and constructed as necessary test constraints to examine and verify the related software artefacts for component correctness. Test contracts are typically realised and represented with special assertion-based test operations. Since contract artefacts may occur at different SCD levels, the contract mapping may also take place to derive test contracts at different mapping levels shown in Figure 8.14 (a) and Figure 8.14 (b). Accordingly, Step TM6 results in system test contracts, component test

contracts, object test contracts, and operation test contracts.



(a) TM6: Mapping Contracts (Diagrammatic Illustration)

	TM6:	Mapping Contracts		
Phase #1:	TM6.1:	system contracts	→	system test contracts
	TM6.2:	component contracts	→	component test contracts
	TM6.3:	object contracts	→	object test contracts
	TM6.4:	operation contracts	→	operation test contracts
Phase #2:	TM6.5:	test contracts	→	test groups <TestGroup>
	TM6.6:	test contracts	→	test operations <TestOperation>
	TM6.7:	test contracts	→	atomic test operations <TestMethod>

(b) TM6: Mapping Contracts (Tabular Illustration)

Figure 8.14 TM6: Mapping Contracts

(1) TM6.1: system contracts → system test contracts

At the system level, a system contract may be applied to the system-level artefact under test, such as a system scenario, event, message or operation. Mapping a system contract may

produce one or more system test contracts to verify the related system artefact.

(2) TM6.2: component contracts → component test contracts

At the component level, a system contract may be applied to the component-level artefact under test, such as a component scenario, message or operation. Mapping a component contract may produce one or more component test contracts to verify the related component artefact.

(3) TM6.3: object contracts → object test contracts

At the object level, an object contract may be applied to the object-level artefact under test, such as an object message or operation. Mapping an object contract may produce one or more object test contracts to verify the related object artefact.

(4) TM6.4: operation contracts → operation test contracts

At the operation level, an operation contract may be applied to the operation under test. Mapping an operation contract may produce one or more operation test contracts to verify the related operation.

Realised with special test operations, test contracts may be additional to the original CUT and are added to constitute relevant test classes and test artefacts. Accordingly, after contracts are mapped out in CTM Phase #1 as described above, a test contract, depending on its complexity, needs to be further mapped to one or more test operations. Following the defined CTM relationship, we need to consider the following mapping cases in CTM Phase #2:

(a) (1 – 1) simple mapping relationship

One test contract is mapped and corresponds to one atomic test operation (represented with XML element <TestMethod> or <TestConstructor>) in SCT_Set. This case often occurs when the test contract is realised and represented with a simple test operation. For example, after we conduct Step TM4.6.4 and Step TM6.7, atomic test operation <TestMethod Name="checkState" …> (as shown in Figure 8.12) represents test contract 2.3 ETC being directed to examine the in-PhotoCell sensor device in the expected state of "IN_PC_OCCUPIED". The test contract also requires checking the associated <Result> of the test operation <TestMethod> to detect whether the related state is correct as expected (as shown in Figure 8.12).

(b) (1 – n) general mapping relationship

One test contract is mapped and corresponds to several atomic test operations (represented with XML element <TestMethod> or <TestConstructor>) in SCT_Set. In this case, one test contract is realised and represented with several tests specified with several test elements. These generated tests are then structured and organised into certain test sequences made up of

related structural elements <TestOperation> or <TestGroup> as necessary. This case may occur when the operation under test causes changes in multiple states, or the corresponding object interaction involves the communication of multiple events. In these situations, multiple test operations are needed to realise and specify a complex test contract in order to examine and verify a complex operation or object interaction, as illustrated with Steps TM6.2 to TM6.5/TM6.6 in Figure 8.14 (a) and Figure 8.14 (b).

(5) TM6.5: test contracts → test groups <TestGroup>

In accordance with the (1 – n) general mapping relationship, a test contract, which is realised and represented with several test operations, is mapped to a CTS test element <TestGroup>, which may further enclose several basic CTS test elements, such as atomic test operations.

(6) TM6.6: test contracts → test operations <TestOperation>

In accordance with the (1 – 1) simple mapping relationship, a test contract, which is realised and represented with a simple test operation, is mapped to a CTS test element <TestOperation>, which may further enclose one or more basic CTS test elements, such as atomic test operations.

(7) TM6.7: test contracts → atomic test operations <TestMethod>

In accordance with the (1 – 1) simple mapping relationship, a test contract, which is realised and represented with a simple test operation, is mapped to a CTS atomic test operation <TestMethod>.

8.3.3 Setting and Applying CTM Criteria

A proven test derivation with any testing technique should conform to certain testing rules or criteria that are able to not only carry out but also regulate component test derivation for the purposes of testing correctness and effectiveness. To ensure the test mapping correctness and quality, we develop certain CTM criteria for effective test mapping and transformation to derive target component test cases. We focus on identifying and designing two main types of CTM criteria: CTM correctness criteria and CTM optimising criteria.

8.3.3.1 CTM Correctness Criteria

The CTM correctness criteria focus on dynamic testing rules, and aim to ensure that component test cases are correctly derived with the CTM technique. One important issue is about test sequences, which organise and structure test artefacts. The sequencing logic is a central focus of sequence mapping because it describes the working procedure and logic of related test artefacts.

At this point, it is useful to be reminded that all software programs on a single processor are executed sequentially, no matter how they are designed and tested. This sequencing execution characteristic is also expressed with UML modeling (which is used for both SCD and SCT in our MBSCT methodology), where UML sequence diagrams clearly illustrate how logically time-ordering interactions (messages/operations) work between participating objects in scenarios for CIT. Therefore, to ensure the sequence mapping correctness, we introduce and define a useful CTM correctness criterion as follows:

CTM Correctness Criteria: sequence consistency matching criterion

The sequence of test messages/operations that contain test elements for constructing component test cases should consistently match (e.g. in the same sequential logical order) the sequence of corresponding interacting messages /operations that are used as the basis to derive the test messages/operations.

Based on this CTM criterion, when some individual testing-related software artefacts are mapped and transformed to become test artefacts, the logic of sequencing order should remain unchanged, that is, the sequencing logic of the test sequence in the SCT context should be consistent with the sequencing logic of the corresponding message/operation sequence in the SCD context. Any mismatch may change the sequencing logic and lead to incompatible or incorrect test derivation, which may contradict test requirements and/or the functional logic of the CUT. Test contracts that are constructed and added into test sequences are also required to be consistent in following the relevant sequencing logic. For example, test group 2.3 TG as shown in Figure 8.12 is a test sequence composed of (i) test operation 2.2 TO, and (ii) test operation 2.3 TO. This test sequence matches with the sequential order as illustrated with the related sequence diagram in Figure 5.4 in Section 5.5. Test contract 2.3 ETC is constructed and added into this test sequence, whose consistent sequence ordering remains unchanged.

The CTM correctness criteria described above can be used as a type of test correctness checking mechanism to examine the relevant rules or constraints applying to test mapping and transformation. Checking test mapping correctness requires conformance to mapping consistency, compliance and compatibility for correct test derivation. During the CTM process, applying the CTM correctness criteria could uncover test mapping problems, and identify possible adjustments needed to regulate the test derivation process. If this occurs, this situation indicates that the prior test design may contain some defects, and test improvement is required to prevent incorrect test cases being derived. This is a useful way that the CTM correctness criteria can aid in identifying test design problems to ensure test mapping correctness for effective test derivation.

8.3.3.2 CTM Optimising Criteria

The CTM optimising criteria focus on static testing and structural rules, and aim to optimise the test mapping and derivation to achieve better test effectiveness. We introduce and define a useful CTM optimising criterion as follows:

CTM Optimising Criteria: sequence formatting/structuring criterion

Test messages/operations and underlying test elements can be structured and optimised to maintain the consistent hierarchical structure and format (e.g. recursive, nested indentation rules at the same logical level) of corresponding interacting messages/operations that occur over time in related sequence diagrams and/or programs for the CUT.

The consistent and uniform structure between test artefacts and component/model artefacts can aid in producing a well-formed structure and format of the target CTS test case specification. A consistent structure also indicates that related test operations work closely together for a specific common testing objective and thus can be organised in the same structured test sequence at the same level. For example, a collection of consecutive test operations in Figure 8.12 jointly work to achieve a common testing objective: examining the composite operation that the in-PhotoCell sensor device senses that the PAL entry point is occupied by the test car and is set to the state of "IN_PC_OCCUPIED". So these test operations and associated test contracts are organised into the same structured test group. In addition, following this CTM criterion, we use leading dot points positioned before each line in Figure 8.12 to highlight certain appropriate structural indentations among the different CTS elements, which emphasises the hierarchical format of the target CTS test case specification.

8.4 Deriving CTS Test Case Specifications

As described in Section 8.2 and Section 8.3, based on test artefacts using UML-based test models and model-based component test design with the MBSCT methodology as shown earlier in Chapter 5 and Chapter 6, we are able to apply the CTM technique to derive target component test cases. Taking the CPS TUC1 test scenario as an illustrative example, the following description summarises how the target CTS test case specification was generated with the MBSCT methodology:

(1) Figure 5.2 showed the developed use case test model and system test scenario.

(2) Figure 5.4 showed the developed design object test model and sub test scenarios.

(3) Figure 6.3 showed the developed overall test sequence with component test design.

(4) Figure 6.4 and Figure 6.5 showed the structured test sequences and test groups in component test design.

(5) Table 6.3 showed all developed test artefacts with component test design, including test sequences, test groups, test operations, test contracts and test states.

(6) Figure 8.6 showed the derived abstract test cases in the form of the sequences of system test events.

(7) Figure 8.7 showed the derived abstract test cases in the form of the sequences of object test messages.

(8) Figure 8.8 showed the derived abstract test cases in the form of the sequences of concrete test operations and associated test contracts.

(9) Figure 8.4 showed the derived overall CTS test sets.

(10) Figure 8.9 showed the derived CTS test sequences (test sets/groups/operations).

(11) Figure 8.12 showed the derived CTS test groups, test operations, test contracts and basic test elements.

Finally, we are able to obtain the full target CTS test case specification generated for the CPS TUC1 test scenario, which is shown in Figure B.5 in Appendix B.

8.5 Summary and Discussion

This chapter has introduced the CTM technique as a new mapping-based test derivation approach and applied it to derive target component test cases in the second phase of the MBSCT framework. We introduced the CTM definition as a unified testing concept for each of the steps of test mapping and transformations that are employed by the CTM technique for model-based component test derivation. We developed the stepwise CTM process with a series of mapping steps to provide practical test transformation strategies and guidance on how to transform model-based test artefacts into abstract test cases and to generate target component test cases. We also developed the useful CTM criteria to ensure test mapping correctness, effectiveness and quality. At the same time, we showed how to apply the CTM technique to component test derivation with the CPS case study. The illustrative examples have demonstrated that, based on test artefacts with UML-based test models and model-based component test design with the MBSCT methodology, the CTM technique is capable of generating target component test cases, such as CTS test case specifications. Therefore, this chapter has demonstrated and validated the

MBSCT testing applicability and capability particularly for component test design and generation (which is the core MBSCT testing capability #2 as described in Section 4.6). This is a major contribution of the CTM technique.

The MBSCT methodology presented in the previous chapters (Chapter 4 to Chapter 7) has showed that our method of component test development holds the five technical characteristics; that is, component test development is model-based, process-based, scenario-based, contract-based and FDD-based. The CTM technique presented in this chapter further enhances our method of component test development, enabling it to have the sixth technical characteristic; that is, component test development is also mapping-based. This is a major feature of the CTM technique.

After showing the MBSCT methodology and its framework in Chapter 4 to Chapter 8, we will undertake more comprehensive methodology validation and evaluation in Chapter 9.

Chapter 9
Methodology Validation and Evaluation

9.1 Introduction

A software testing approach needs to be properly validated and evaluated before it can be adopted in the testing practice. The previous chapters of this thesis (Chapter 4 to Chapter 8) have presented the MBSCT methodology and its framework developed by this research. At the same time, many illustrative examples have been used to demonstrate how to apply the MBSCT methodology and its framework to undertake UML-based SCT, particularly test model construction, model-based component test design and generation, and component fault detection, diagnosis and localisation. Based on this, this chapter specifically undertakes further methodology validation by evaluating the MBSCT testing capabilities with more comprehensive case studies.

This chapter reports a series of two full case studies undertaken in this research. Section 9.2 describes an overview of case study design and setup. Section 9.3 presents the first core case study, the Car Parking System. Section 9.4 presents the second major case study, the Automated Teller Machine system. Section 9.5 conducts evaluation comparison and discussions on case studies. Section 9.6 summarises this chapter.

9.2 Case Study Design

We employ case studies to carry out methodology validation and evaluation, because case study research is known as an effective empirical study method in software engineering [86] [111] [75] [126] [161]. This section describes an overview of case study design and setup.

The major objectives of our case studies are to validate and evaluate the six core MBSCT testing capabilities of principal interest (as described earlier in Section 4.6). Our objectives are described as follows:

(1) Evaluating the *testing applicability* of the MBSCT methodology

We carry out case studies to demonstrate and validate that the MBSCT methodology and its framework can be practically applied to UML-based CIT (Component Integration Testing). This is measured in terms of primary MBSCT capabilities in the following three aspects:

(a) MBSCT Capability #1: test model construction

(b) MBSCT Capability #2: component test design and generation

(c) MBSCT Capability #3: component fault detection, diagnosis and localisation

(2) Evaluating the *testing effectiveness* of the MBSCT methodology

We carry out case studies to validate and evaluate the MBSCT testing effectiveness, which is measured in terms of the following important MBSCT testing capabilities:

(a) MBSCT Capability #4: adequate test artefact coverage

We validate and evaluate the MBSCT methodology that is capable of achieving adequate test artefact coverage of testing-related component/model artefacts and associated test contracts for the purpose of effective model-based component testing.

(b) MBSCT Capability #5: component testability improvement

We validate and evaluate the MBSCT methodology that is capable of bridging the identified "test gaps" and improving component testability effectively for fulfilling testing requirements.

(c) MBSCT Capability #6: adequate component fault coverage and diagnostic solutions

Validation and evaluation of the FDD capability is regarded as a major method for assessing the effectiveness of software testing approaches [33] [37]. We validate and evaluate the MBSCT methodology that is capable of achieving adequate component fault coverage and diagnostic solutions for the purpose of effective FDD and fulfilling testing requirements.

Our case studies are designed following the generally accepted structure of empirical study methods in software engineering (as referred to above). The *object* of each case study is the MBSCT methodology and its framework. The *quality focus* of each case study is the applicability and effectiveness of the MBSCT methodology and its framework for UML-based SCT. The *perspective* of each case study is from the viewpoint of software testing researchers. The *context* of each case study is this research project. The *subject* of each case study is the researcher. Each case study was performed off-line in an academic research environment (i.e. non-industry software development). The scope for each case study was limited to a single-object study by a single subject because of the constraint of available research resources and time.

Two full case studies have been carried out. Each case study was conducted in the following six main steps:

(1) Step #1: Constructing test models (Capability #1)
(2) Step #2: Designing and generating component tests (Capability #2)
(3) Step #3: Evaluating test artefact coverage and adequacy (Capability #4)
(4) Step #4: Evaluating component testability improvement (Capability #5)
(5) Step #5: Detecting, diagnosing and locating component faults (Capability #3)

(6) Step #6: Evaluating component fault coverage and diagnostic solutions (Capability #6)

Our case study description is structured in terms of the important tasks of testing and evaluation undertaken in the above main steps. By using many test evaluation examples selected from the CPS TUC1 test scenario, the previous chapters (Chapter 4 to Chapter 8) have systematically illustrated and demonstrated how to apply the MBSCT methodology and its framework to UML-based SCT activities, with a specific emphasis on the validation of the MBSCT testing applicability (including the core MBSCT testing capabilities #1 to #3). On this basis, the two case studies presented in this chapter particularly focus on validating and evaluating the MBSCT testing effectiveness (including the core MBSCT testing capabilities #4 to #6).

9.3 Case Study: Car Parking System

The first core case study is the testing of the Car Parking System (CPS) undertaken in this research. In order to further validate and evaluate the core MBSCT testing capabilities, the full CPS case study has been undertaken to exercise and examine all three CPS TUC core test scenarios (including TUC1, TUC2 and TUC3 in the three major parking phases), which constitute an overall test scenario/sequence of one full parking access process cycle for any parking car. This section reports important testing aspects and evaluation results of the CPS case study, with respect to adequate test artefact coverage (in Section 9.3.2), component testability improvement (in Section 9.3.3), fault case scenario analysis and diagnostic solution design (in Section 9.3.4), adequate component fault coverage and fault diagnostic solutions and results (in Section 9.3.5). Other relevant testing aspects (such as test model development, model-based component test design and generation, etc.) and evaluation results are included in Appendix B. The full CPS case study has been described earlier in [168] [170].

9.3.1 Special Testing Requirements

This section describes important special testing requirements for testing the CPS system. In addition to the usual system operations and functional requirements as described in Appendix B, we have identified and examined a set of special quality requirements for supporting secure and reliable parking services, which are the principal focus of testing and evaluation conducted in the CPS case study. By using a series of three illustrative test evaluation examples (#1, #2, and #3), the CPS case study was undertaken to particularly demonstrate and evaluate how the core MBSCT testing capabilities can be effectively applied to test the CPS system to fulfil the three most important CPS special testing requirements (#1, #2, and #3).

In this chapter, we describe one selected CPS special testing requirement #1 in Section 9.3.1, and relevant testing aspects and evaluation results with the evaluation example #1 particularly in association with this special testing requirement in subsequent Sections 9.3.2 to 9.3.5. Appendix B includes all three CPS special testing requirements (in Section B.2), and shows relevant testing aspects and evaluation results with the evaluation examples #2 and #3 (especially in Sections B.6 to B.8) for the other two CPS special testing requirements #2 and #3.

As an illustrative example, we select "Special Testing Requirement #1: Parking Access Safety Rule", which is specified as follows:

(1) Special Testing Requirement #1: Parking Access Safety Rule

In the CPS system, all parking cars must abide by the parking access safety rule – "*one access at a time*", with the following specific mandatory public access requirements:

(a) Only one car can access the PAL (Parking Access Lane) at a time. This means that it is not allowed that two or more cars access the PAL at any same time.

(b) The next car is allowed to access the PAL only after the last car has finished its full PAL access.

This CPS safety rule is jointly supported by the correct control operations of the Traffic Light device and the In-PhotoCell Sensor device operated at the PAL entry point. This rule can prevent the occurrences of unsafe scenarios, e.g. possible car collisions due to multiple concurrent car accesses.

9.3.2 Evaluating Test Artefact Coverage and Adequacy

This section analyses test artefacts derived with the MBSCT methodology in the CPS case study, and evaluates how they are able to achieve adequate test artefact coverage for the CIT purpose. This test evaluation aspect is also further discussed with the testability evaluation in the next Section 9.3.3.

In Appendix B for the CPS case study, Section B.4 shows that the constructed CPS test models cover sufficient testing-required model artefacts and corresponding component/object operations that participate in SCI in test scenarios (as illustrated in Figure B.2, Table B.1 and Figure B.3). Section B.5 shows that the CPS test sequence design covers all testing-required parking control operations of the associated CPS control devices and car movements along the PAL (as illustrated in Figure B.4), and that the CPS component test design provides the defined test data for all the covered test artefacts (as illustrated in Table B.2). Adequate test artefact coverage is technically supported by the TbC test contract criteria of the TbC technique.

In the CPS case study, the evaluation of test artefact coverage and adequacy can be measured in terms of the number of different types of test artefacts used for the CPS test design, which is shown in Table 9.1. We can observe that there were a total of three (3) main test scenarios/sequences, a total of eight (8) sub test scenarios/sequences, a total of eighteen (18) test groups, a total of twenty-three (23) test operations, a total of eighteen (18) test contracts, and a total of ten (10) (different) test states in the CPS component test design.

Table 9.1 Measurement of Test Artefact Coverage (CPS Case Study)

	Test Scenario	No. of Test Sequences	No. of Test Groups	No. of Test Operations	No. of Test Contracts	No. of Test States
	CPS TUC1	3	7	9	7	4 + 1 (6)
	CPS TUC2	2	4	5	4	2 + 1 (4)
	CPS TUC3	3	7	9	7	4 + 1 (7)
Total	3	8	18	23	18	10 + 3 (17)

Note that, among the seventeen (17) test states being used in component test design, there were only ten (10) different test states, which correspond to ten (10) individual CPS control states. There were three (3) special test states (including "SB_DOWN", "TL_RED", "TD_WITHDRAWN") that are repeatedly used in the preconditions/postconditions between the boundaries of the three CPS TUC test scenarios respectively as follows:

(a) The special test state of "TL_RED" is used in the postcondition of the current CPS TUC1 test scenario and also in the precondition of the next CPS TUC2 test scenario;

(b) The special test state of "TD_WITHDRAWN" is used in the postcondition of the current CPS TUC2 test scenario and also in the precondition of the next CPS TUC3 test scenario;

(c) The special test state of "SB_DOWN" is used in the postcondition of the current CPS TUC3 test scenario and also in the precondition of the next CPS TUC1 test scenario.

9.3.3 Evaluating Component Testability Improvement

Adequate test artefact coverage creates a solid foundation for achieving good model-based component testability improvement. The testability improvement is fulfilled by applying the MBSCT methodology (especially the two MBSCT methodological components: the TbC technique and the TCR strategy) to test model construction and contract-based test design, as described earlier in Chapter 4 to Chapter 7. These chapters have demonstrated the MBSCT testing

capabilities to improve component testability by means of bridging the previously identified "test gaps" (including both *Test-Gap #1* and *Test-Gap #2*, as described in Section 5.2.4.2). These chapters have illustrated and discussed many testing examples in detail for the CPS TUC1 test scenario, which technically paves the way for our further evaluation in this section.

We further examine and evaluate the effectiveness of the MBSCT testing capabilities #4 and #5 (as described in Section 9.2) across all three CPS TUC core test scenarios in the CPS case study. In particular, we illustrate the three relevant evaluation examples with the CPS component test design, and evaluate how adequate test artefact coverage and component testability improvement can be achieved to fulfil the three CPS special testing requirements. As an illustrative evaluation example, the next Section 9.3.3.1 presents "Evaluation Example #1: Parking Access Safety Rule" (for the first CPS special testing requirement). Another two evaluation examples #2 and #3 (for the two CPS special testing requirements #2 and #3) are included in Section B.6 in Appendix B. Then, Section 9.3.3.2 presents an evaluation summary with the three evaluation examples.

9.3.3.1 Evaluation Example #1: Parking Access Safety Rule

This section presents the first evaluation example, which is about the CPS special testing requirement #1 (Parking Access Safety Rule) and is related to the testing of the traffic light device in the CPS TUC1 test scenario. Because the control operations of the traffic light device are exercised and examined in the CPS TUC1 integration testing context, the testing is CIT-related.

The CPS system has a special testing requirement of the "*one access at a time*" rule for the mandatory public access safety purpose (as described in Section 9.3.1). The testing of this CPS safety rule requires sufficient test coverage for exercising and examining the testing-required control operations of the traffic light device, and the main test operations include 1.2 TO `setGreen`() and 3.2 TO `setRed`() in the CPS TUC1 test scenario. As described in Section B.5 in Appendix B and Section 9.3.2 above, the CPS test sequence design and component test design undertaken in the CPS case study have provided adequate test artefact coverage for this testing requirement, which bridges *Test-Gap #1*. In addition, the CPS component test design constructs and applies appropriate test contracts to each of these testing-required control operations for testing the traffic light device. The main test contracts comprise 1.2 ITC `checkState`(`trafficLight`, "TL_GREEN") and 3.2 ITC `checkState`(`trafficLight`, "TL_RED"), which improve component testability by enabling testing to evaluate relevant test results and so bridges *Test-Gap #2*. Therefore, the CPS component test design can effectively improve component testability and fulfil the CPS special testing requirement #1.

9.3.3.2 Evaluation Summary: Adequate Test Artefact Coverage and Component Testability Improvement

Based on the three evaluation examples and relevant discussions for the MBSCT evaluation (as described in Section 9.3.2 and Section 9.3.3.1 above, and Section B.6.1 and Section B.6.2 in Appendix B), the evaluation of adequate test artefact coverage and component testability improvement with the CPS case study can be summarised as shown as in Table 9.2. This table shows three main evaluation result sets (in three rows) that are assessed in terms of test scenarios, adequate test artefact coverage, testability improvement (i.e. bridging the "test gaps", including both *Test-Gap #1* and *Test-Gap #2*), and testing requirement fulfilment.

Table 9.2 Evaluation Summary: Adequate Test Artefact Coverage and Component Testability Improvement (CPS Case Study)

Special Testing Requirement	Test Scenario	Adequate Test Artefact Coverage	Testability Improvement		Testing Requirement Fulfilment
			Bridging Test-Gap #1	Bridging Test-Gap #2	
#1: Parking Access Safety Rule	CPS TUC1	Yes	Yes	Yes	Yes
#2: Parking Pay-Service Rule	CPS TUC2	Yes	Yes	Yes	Yes
#3: Parking Service Security Rule	CPS TUC3	Yes	Yes	Yes	Yes

Our evaluation has concluded the following important points:

(1) Based on the relevant evaluation as described in Section 9.3.2 and Section 9.3.3.1 above, the first evaluation result set has drawn the conclusion that the CPS component test design in the CPS TUC1 test scenario is capable of achieving adequate test artefact coverage, improving component testability and fulfilling the CPS special testing requirement #1: Parking Access Safety Rule.

(2) Based on the relevant evaluation in Section 9.3.2 above and Section B.6.1 in Appendix B, the second evaluation result set has drawn the conclusion that the CPS component test design in the CPS TUC2 test scenario is capable of achieving adequate test artefact coverage, improving component testability and fulfilling the CPS special testing requirement #2: Parking Pay-Service Rule.

(3) Based on the relevant evaluation in Section 9.3.2 above and Section B.6.2 in Appendix B,

the third evaluation result set has drawn the conclusion that the CPS component test design in the CPS TUC3 test scenario is capable of achieving adequate test artefact coverage, improving component testability and fulfilling the CPS special testing requirement #3: Parking Service Security Rule.

(4) Finally, our evaluation concludes that the CPS component test design with the MBSCT methodology can fulfil the three CPS special testing requirements for effective testing of the CPS system, and the effectiveness of the MBSCT testing capabilities #4 and #5 (for adequate test artefact coverage and component testability improvement) can be achieved as required.

9.3.4 Detecting, Diagnosing and Locating Component Faults

Validating and evaluating fault diagnosis capability is commonly used as a key approach to the assessment of the effectiveness of SCT methods (as indicated earlier in Section 9.2 (2) (c)). Adequate test artefact coverage and testability improvement jointly create a solid foundation for component fault detection, diagnosis and localisation. This is accomplished effectively by applying the TbC technique (especially, the CBFDD method) to FDD activities, as described earlier in Chapter 7, where we have demonstrated the MBSCT testing capability of not only fault detection, but also fault diagnosis to locate component faults for correction or removal. Chapter 7 also described many relevant illustrative FDD examples for the CPS TUC1 test scenario.

On this basis, we further examine and evaluate the MBSCT testing capabilities #3 and #6 (as described in Section 9.2) for fault detection, diagnosis and localisation with the CPS case study. Consistent with the FDD activities as discussed earlier in Chapter 7, we examine the actual CPS integration-level faults (e.g. which cause certain major CPS integration fault/failure scenarios) to fulfil the three CPS special testing requirements. Specifically, we demonstrate three illustrative FDD evaluation examples for fault case scenario analysis and fault diagnostic solution design: the first evaluation example for “Evaluation Example #1: Parking Access Safety Rule” (for the first CPS special testing requirement) is presented in Section 9.3.4.1 below, and two other evaluation examples #2 and #3 (for the two CPS special testing requirements #2 and #3) are described in Section B.7 in Appendix B.

Each FDD example is described in the following six main parts:

(1) Fault Case Scenario and Analysis: describing what the fault is about (especially major fault/failure scenarios) against a specific CPS special testing requirement in the CPS system. The fault is to be detected, diagnosed and located with the two types of fault diagnostic solutions that are described below.

(2) Fault-Related Test Scenario: indicating which CPS TUC test scenario is related to the fault under diagnosis, and this related test scenario must cover the fault case scenario.

(3) Fault-Related Control Point: indicating which main CPS control point (e.g. the entry point, the ticket point or the exit point in the PAL) is related to the fault under diagnosis, and this control point is where the fault occurs.

(4) Fault-Related Control Device: indicating which CPS control device is related to the fault under diagnosis, and this control device operating at the fault-related control point is the cause of the fault.

(5) Direct Diagnostic Solution: A fault diagnostic solution that is obtained with the CBFDD method, based on the relevant information of component design and/or certain testing-support features (especially as described earlier in Section 7.6.2.2).

(6) Stepwise Diagnostic Solution: A fault diagnostic solution that is obtained with the CBFDD method, especially by applying the stepwise CBFDD guidelines (as described earlier in Section 7.5.5 and Section 7.6.2.3). Note that these two types of fault diagnostic solutions are equivalent for diagnosing and locating the same fault, as discussed earlier in Section 7.6.2.3.3. In the following (especially in Section 9.3.5 onwards), our FDD descriptions mainly focus on direct diagnostic solutions.

9.3.4.1 Evaluation Example #1: Parking Access Safety Rule

(1) Fault Case Scenario And Analysis

For the major fault/failure scenario of the CPS safety rule: while the current car enters the PAL entry point but has not finished its full PAL access yet, another unauthorised car illegally enters and accesses the PAL at the same time. The resulting failure is a safety violation of the "*one access at a time*" rule against the CPS special testing requirement #1.

(2) Fault-Related Test Scenario

This fault is related to the CPS TUC1 test scenario, where the fault diagnosis is CIT-related.

(3) Fault-Related Control Point

This fault is related to the CPS control point – the entry point in the PAL.

(4) Fault-Related Control Device

This fault is related to the CPS control device – the traffic light device, which is operated

at the PAL entry point.

(5) Direct Diagnostic Solution

The fault diagnostic solution for the CPS test design is to incorporate the following test groups in the CPS TUC1 test scenario:

(a) Test group 1.2 TG contains test operation 1.2 TO `setGreen`() and its associated (post-condition) test contract 1.2 ITC `checkState`(`trafficLight`, "TL_GREEN"), and test state "TL_GREEN".

(b) Test group 3.2 TG contains test operation 3.2 TO `setRed`() and its associated (postcondition) test contract 3.2 ITC `checkState`(`trafficLight`, "TL_RED"), and test state "TL_RED".

(6) Stepwise Diagnostic Solution

The fault diagnostic solution for the CPS TUC1 test design is to incorporate the following equivalent test artefacts as a special test group:

(a) Precondition: test contract TC_TL_GREEN, which functions equivalently to test contract 1.2 ITC in test group 1.2 TG in the CPS TUC1 test scenario.

(b) Test operation TO_TL_RED, which functions equivalently to test operation 3.2 TO in test group 3.2 TG in the CPS TUC1 test scenario.

(c) Postcondition: test contract TC_TL_RED, which functions equivalently to test contract 3.2 ITC in test group 3.2 TG in the CPS TUC1 test scenario.

9.3.5 Evaluating Adequate Component Fault Coverage and Diagnostic Solutions

Based on the relevant discussions about the MBSCT assessment in Section 9.3.2 to Section 9.3.4 and Section B.4 to Section B.7 in Appendix B (especially for fault case scenario analysis and diagnostic solution design), we undertake further examination and evaluation of the effectiveness of the MBSCT testing capability #6 for adequate component fault coverage (in Section 9.3.5.1), fault diagnostic solutions and results (in Section 9.3.5.2).

9.3.5.1 Adequate Component Fault Coverage

The MBSCT methodology employs test groups as the primary mechanism to achieve adequate component fault coverage. In particular, at least one *basic test group* (usually consisting of at least a test operation and its associated test contract as well as relevant test states) is used to cover and diagnose a possible fault related to the component/object operation under test. Such basic test groups can be regarded as basic test cases, which form the primary testing basis for

developing *basic fault diagnostic solutions* (consisting of one or more basic test groups) and component integration test cases.

In the following, we analyse and evaluate adequate component fault coverage for fault diagnosis of the CPS system:

(1) The CPS system comprises the five (5) main individual control devices (including traffic light, in-PhotoCell sensor, ticket dispenser, stopping bar, and out-PhotoCell sensor), which are located at the three (3) main control points (i.e. entry point, ticket point and exit point) along the PAL.

(2) Based on the contractual rules and relationships for the normal CPS operation, a CPS control device works only in the two (2) main correct control states.

For example, the traffic light device functions only in the two (2) main correct control states: "TL_GREEN" and "TL_RED", which are *orthogonal* and occur alternatively. These two correct control states of the traffic light device are independent of other CPS control device operations, i.e. their occurrences are not affected by the operation of another CPS control device. Except for these two correct control states, there should be no any other valid control state for the traffic light device at any time in the CPS system.

(3) There are only two (2) possible values related to one individual control state of a CPS control device.

For example, for the CPS control state of "TL_GREEN" of the traffic light device, there are only two (2) possible control state values as follows:

(a) The correct state with the valid state value for the correct control operation, e.g. TL_GREEN.

(b) The incorrect state with some invalid state value for the faulty/incorrect control operation, e.g. its opposite/orthogonal state value of "TL_RED" or any other invalid state value.

(4) Accordingly, a CPS control device can have a total of four (i.e. 2 * 2) possibly-combined control state values. Then, the number of the possibly-combined incorrect control state values

= (the number of the total combinations of all the possible control state values)

– (the only two correct combinations of the two correct control state values)

= 4 – 2 = 2

In other words, a CPS control device may have at least two (2) primary faults. The *primary faults* of a CPS control device are independent of other CPS control device operations. These primary faults may occur both at the unit level and at the integration level.

(5) Therefore, the CPS system may contain a maximum of 10 primary faults (i.e. 2 primary faults/device * 5 devices). These 10 CPS primary faults could occur independently of each other, possibly at both the unit level and the integration level.

(6) Note that these 10 CPS primary faults may also be interrelated, which means that one fault may have resulted from the occurrence of another fault. For example, a typical case is that a preceding fault may cause a violated precondition and then lead to the occurrence of a related succeeding fault in certain execution paths. This indicates that it is necessary to diagnose relevant interrelated faults in order to find all possibly-combined faults (such relevant fault diagnosis is further discussed in the next Section 9.3.5.2).

Based on the above fault coverage analysis and the relevant MBSCT validation and evaluation as discussed in Section 9.3.2 to Section 9.3.4 and Section B.4 to Section B.7 in Appendix B, we can describe a comprehensive analysis and evaluation by using Table 9.3. This table is structured in terms of primary faults, fault case scenario and analysis, and fault coverage with appropriate fault diagnostic solutions, and shows that each primary fault can be adequately covered and diagnosed by at least a basic fault diagnostic solution to fulfil a relevant specific CPS special testing requirement. A *usual* (or commonly-used) *fault diagnostic solution* can combine the two test groups related to the same CPS control device, or some more test groups related to the different CPS control devices, when these test groups and their associated test artefacts are related to the particular CPS primary fault under diagnosis. Therefore, the MBSCT methodology can develop effective test groups and fault diagnostic solutions to adequately cover and diagnose all 10 primary faults in the CPS system to fulfil the three CPS special testing requirements.

Table 9.3 Analysis and Evaluation of Adequate Component Fault Coverage and Diagnostic Solutions (CPS Case Study)

Primary Fault	Fault Case Scenario and Analysis	Control Device	Control Point	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
Special Testing Requirement #1: Parking Access Safety Rule					
1.1 FAULT_TL_GREEN The traffic light device is NOT in the correct control state of "TL_GREEN" as expected.	*Scenario #1*: The next waiting car could not enter the PAL, even after the last car has finished its full PAL access or even though no car is accessing the PAL. This fault may cause that the CPS services could become inaccessible. *Scenario #2*: The test car illegally enters the PAL entry point, even though the test car is not allowed for access permission. This fault may cause a violated precondition for the related succeeding CPS operation for the current car.	The Traffic Light device	The CPS entry point	The CPS TUC1 test scenario	CPS TUC1 Test Design: Test group 1.2 TG contains test operation 1.2 TO `setGreen`() and its associated (postcondition) test contract 1.2 ITC `checkState`(`trafficLight`, "TL_GREEN").
1.2 FAULT_TL_RED The traffic light device is NOT in the correct control state of "TL_RED" as expected.	While the current car enters the PAL entry point but has not finished its full PAL access yet, another unauthorised car illegally enters and accesses the PAL at the same time. The resulting failure is a safety violation of the "*one access at a time*" rule against the CPS special testing requirement #1.	The Traffic Light device	The CPS entry point	The CPS TUC1 test scenario	CPS TUC1 Test Design: Test group 3.2 TG contains test operation 3.2 TO `setRed`() and its associated (postcondition) test contract 3.2 ETC `checkState`(`trafficLight`, "TL_RED").
2.1 FAULT_IN_PC_OCCUPIED The in-PhotoCell sensor device is NOT in the correct control state of "IN_PC_OCCUPIED" as expected.	The in-PhotoCell sensor device fails to sense that the PAL entry point has been occupied by the entering car (i.e. the test car is accessing the PAL entry point). This fault may cause a violated precondition for the related succeeding CPS operation for the current car, or a failure that the CPS entry point becomes inaccessible.	The In-PhotoCell Sensor device	The CPS entry point	The CPS TUC1 test scenario	CPS TUC1 Test Design: Test group 2.3 TG contains test operation 2.2 TO `goTo`(`gopace-cross-inPC`, `int`), test operation 2.3 TO `occupy`() and its associated (postcondition) test contract 2.3 ETC `checkState`(`inPhotoCell`, "IN_PC_OCCUPIED").

Primary Fault	Fault Case Scenario and Analysis	Control Device	Control Point	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
2.2 FAULT_IN_PC_CLEARED The in-PhotoCell sensor device is NOT in the correct control state of "IN_PC_CLEARED" as expected.	The in-PhotoCell sensor device fails to sense that the PAL entry point has been cleared by the entering car (i.e. the test car has finished accessing the PAL entry point). This fault could lead to a violated precondition for the related succeeding CPS operation for the current car, or a failure that the CPS entry point is not to be assessable by the next entering car.	The In–PhotoCell Sensor device	The CPS entry point	The CPS TUC1 test scenario	CPS TUC1 Test Design: Test group 2.5 TG contains test operation 2.4 TO `goTo( gopace-crossover-inPC, int )`, test operation 2.5 TO `clear`() and its associated (postcondition) test contract 2.5 ETC `checkState( inPhotoCell`, "IN_PC_CLEARED").
Special Testing Requirement #2: Parking Pay-Service Rule					
3.1 FAULT_TD_DELIVERED The ticket dispenser device is NOT in the correct control state of "TD_DELIVERED" as expected.	The ticket dispenser fails to deliver a ticket to be withdrawn by the test driver. This fault may cause that the test driver could not withdraw the ticket for paying parking fare as expected. The resulting failure could further cause a pay-service violation of the "*no pay, no parking*" rule.	The Ticket Dispenser device	The CPS ticket point	The CPS TUC2 test scenario	CPS TUC2 Test Design: Test group 1.2 TG contains test operation 1.2 TO `deliver`() and its associated (postcondition) test contract 1.2 ITC `checkState( ticketDispenser`, "TD_DELIVERED").
3.2 FAULT_TD_WITHDRAWN The ticket dispenser device is NOT in the correct control state of "TD_WITHDRAWN" as expected.	The test car crosses over the ticket point to move forward towards the PAL exit point, even though the test driver has not withdrawn the ticket for paying parking fare. The resulting failure is a pay-service violation of the "*no pay, no parking*" rule against the CPS special testing requirement #2.	The Ticket Dispenser device	The CPS ticket point	The CPS TUC2 test scenario	CPS TUC2 Test Design: Test group 2.3 TG contains 2.2 TO `goTo( gopace-goto-TD, int )`, test operation 2.3 TO `withdraw`() and its associated (postcondition) test contract 2.3 ETC `checkState( ticketDispenser`, "TD_WITHDRAWN").

Primary Fault	Fault Case Scenario and Analysis	Control Device	Control Point	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
Special Testing Requirement #3: Parking Service Security Rule					
4.1 FAULT_SB_UP The stopping bar device is NOT in the correct control state of "SB_UP" as expected.	The test car cannot go to cross over the PAL exit point to complete its full access to the PAL. This fault may cause a violated precondition for the related succeeding CPS operation for the current car, or a failure that the PAL exit point could become inaccessible (i.e. the current car could not exit the PAL).	The Stopping Bar device	The CPS exit point	The CPS TUC3 test scenario	CPS TUC3 Test Design: Test group 1.2 TG contains test operation 1.2 TO `raise`() and its associated (postcondition) test contract 1.2 ITC `checkState`(`stoppingBar`, "SB_UP").
4.2 FAULT_SB_DOWN The stopping bar device is NOT in the correct control state of "SB_DOWN" as expected.	The stopping bar remains un-lowered (e.g. the stopping bar is still raised to up), even after the current car has finished its full access to the PAL (which means that the current car has already finished accessing the PAL exit point), or even if no car is accessing the PAL. The resulting failure is a security violation of the "*public security protection and maintenance*" rule against the CPS special testing requirement #3.	The Stopping Bar device	The CPS exit point	The CPS TUC3 test scenario	CPS TUC3 Test Design: Test group 3.2 TG contains test operation 3.2 TO `lower`() and its associated (postcondition) test contract 3.2 ITC `checkState`(`stoppingBar`, "SB_DOWN").
5.1 FAULT_OUT_PC_OCCUPIED The out-PhotoCell sensor device is NOT in the correct control state of "OUT_PC_OCCUPIED" as expected.	The out-PhotoCell sensor device fails to sense that the PAL exit point has been occupied by the test car (i.e. the test car is accessing the PAL exit point). This fault may cause a violated precondition for the related succeeding CPS operation for the current car, or a failure that the CPS exit point becomes inaccessible.	The Out-PhotoCell Sensor device	The CPS exit point	The CPS TUC3 test scenario	CPS TUC3 Test Design: Test group 2.3 TG contains test operation 2.2 TO `goTo`(`gopace-cross-outPC`, `int`), test operation 2.3 TO `occupy`() and its associated (postcondition) test contract 2.3 ETC `checkState`(`outPhotoCell`, "OUT_PC_OCCUPIED").
5.2 FAULT_OUT_PC_CLEARED The out-PhotoCell sensor device is NOT in the correct control state of "OUT_PC_CLEARED" as expected.	The out-PhotoCell sensor device fails to sense that the PAL exit point has been cleared by the exiting car (i.e. the test car has finished accessing the PAL exit point). This fault could lead to a violated precondition for the related succeeding CPS operation for the current car, or a failure that the CPS exit point is not to be accessible by the next accessing car.	The Out-PhotoCell Sensor device	The CPS exit point	The CPS TUC3 test scenario	CPS TUC3 Test Design: Test group 2.5 TG contains test operation 2.4 TO `goTo`(`gopace-crossover-outPC`, `int`), test operation 2.5 TO `clear`() and its associated (postcondition) test contract 2.5 ETC `checkState`(`outPhotoCell`, "IN_PC_CLEARED").

9.3.5.2 Fault Diagnostic Solutions: Diagnosis Results and Analysis

Section 9.3.4 and Section 9.3.5.1 have assessed the effectiveness of the MBSCT testing capability for fault case scenario analysis and diagnostic solution design, and adequate component fault coverage. On this basis, this section conducts a more comprehensive examination of our fault diagnostic solutions and their results to further evaluate the MBSCT fault diagnosis capability.

With the MBSCT methodology, test sequences are a core part of component test design (as described in Section 6.5.1 and Section B.5.1 in Appendix B) to develop fault diagnostic solutions. A test sequence comprises an expected execution sequence of component/object operations, where a typical case of interrelated faults may exist: a fault of a preceding operation may trigger and/or produce a violated precondition for a directly/indirectly succeeding operation. Accordingly, this violated precondition could cause the related succeeding operation to be prevented from executing or its execution to fail. This is a useful fault diagnostic feature that can facilitate diagnosing possible interrelated faults.

In particular, based on this feature, we can apply the following *fault diagnostic strategy* to develop useful fault diagnostic solutions for uncovering faults that may cause the same fault/failure case scenario:

(a) When diagnosing the possible faults related to the current operation, it is necessary to exercise and examine its preceding operations that are closely related to its preconditions. The faults of these preceding operations (if they exist) may produce an intermediate error, which, by propagation, could subsequently result in the execution failure of the current operation under diagnosis. This fault diagnostic strategy conforms to the principle of the "*fault causality chain*" as described earlier in Section 7.2.

(b) Accordingly, when developing possible fault diagnostic solutions for diagnosing the possible faults related to the current operation under diagnosis, we can apply this fault diagnosis strategy to conduct fault diagnosis of its preceding operations. Note that such preceding operations include the immediately preceding operation just before the current operation and other non-immediately preceding operations, and these preceding operations' execution may affect some precondition of the execution of the current operation.

(c) To diagnose the possible faults causing the same fault/failure case scenario, a fault with the current operation is a *directly-related fault* causing this fault/failure case scenario. In addition, a fault with a preceding operation is an *indirectly-related fault* that could result in the occurrence of the same fault/failure case scenario. Usually for the same fault/failure case scenario, there may be more than one indirectly-related faults, but there is only one directly-related primary fault that is associated with the current operation under diagnosis.

Effective fault diagnostic solutions must be able to cover and diagnose all possible directly and indirectly related faults to achieve the desired fault diagnosis capability. In the CPS case study, we describe the three illustrative FDD evaluation examples using our fault diagnosis strategy (as described above) and our fault diagnostic solutions (as illustrated in Table 9.3) to detect, diagnose and locate the possible directly and indirectly related faults that violate the three CPS special testing requirements. The next Section 9.3.5.2.1 describes "Evaluation Example #1: Parking Access Safety Rule" (for the first CPS special testing requirement). Another two evaluation examples #2 and #3 (for the two CPS special testing requirements #2 and #3) are shown in Section B.8 in Appendix B. Then, Section 9.3.5.3 presents a FDD evaluation summary with the three evaluation examples.

9.3.5.2.1 Evaluation Example #1: Parking Access Safety Rule

This subsection diagnoses the possible directly and indirectly related faults causing the major failure scenario of the CPS safety rule against the CPS special testing requirement #1. In the CPS case study, we developed and applied three individual fault diagnostic solutions (as described in Section 9.3.4.1 and Table 9.3 above). Each fault diagnostic solution incorporated the relevant test groups in the CPS TUC1 test scenario for the CPS test design (as illustrated in Figure 9.1 below).

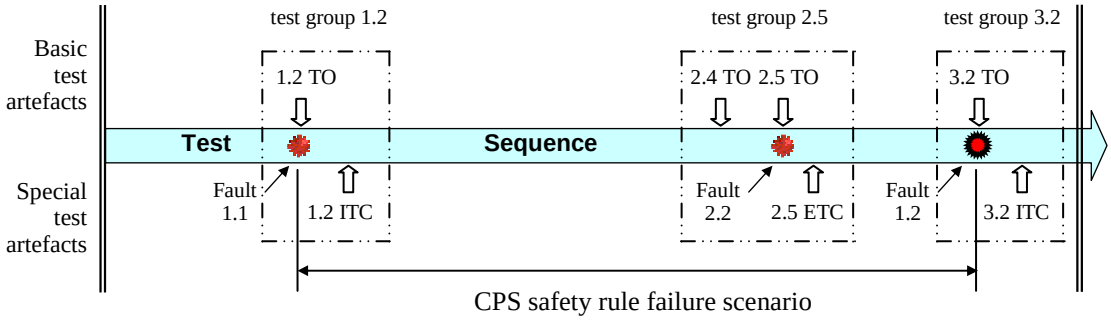


Figure 9.1 Evaluation Example #1: Parking Access Safety Rule
(Fault Diagnostic Solutions with the CPS TUC1 Test Design)

Our FDD evaluation for this major fault/failure scenario is described as follows:

(1) Primary Fault 1.2 FAULT_TL_RED (as described in Table 9.3)

For diagnosing the directly-related primary fault, the first fault diagnostic solution we developed is that the CPS TUC1 test design employs test group 3.2 TG to exercise test operation 3.2 TO `setRed`(), which is verified by its associated (postcondition) test contract 3.2 ITC

checkState(trafficLight, “TL_RED”) and test state “TL_RED” in the CPS TUC1 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed the following fault: the fault is related to the traffic light device operated at the PAL entry point, where this CPS device fails in the execution of operation setRed(), causing the traffic light device NOT to be in the correct control state of “TL_RED” as expected. This is Primary Fault 1.2 FAULT_TL_RED as described in Table 9.3, which leads to a failure to maintain the CPS safety rule (“*one access at a time*”) against the CPS special testing requirement #1.

Therefore, Primary Fault 1.2 FAULT_TL_RED directly causes the major fault/failure scenario of the CPS safety rule as described in Section 9.3.4.1. The first fault diagnostic solution is able to diagnose this directly-related primary fault. Following Step #6 of the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related operation setRed() of the traffic light device (as illustrated earlier in Step #6 in Section 7.6.2.3.2).

(2) Primary Fault 1.1 FAULT_TL_GREEN (as described in Table 9.3)

To diagnose an indirectly-related primary fault, the second fault diagnostic solution we developed is that the CPS TUC1 test design uses test group 1.2 TG to exercise test operation 1.2 TO setGreen(), which is verified by its associated (postcondition) test contract 1.2 ITC checkState(trafficLight, “TL_GREEN”) and test state “TL_GREEN” in the CPS TUC1 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed a fault: the fault is related to the traffic light device operated at the PAL entry point, where the traffic light device fails in the execution of operation setGreen(), causing the traffic light device NOT to be in the correct control state of “TL_GREEN” as expected. This is Primary Fault 1.1 FAULT_TL_GREEN as described in Table 9.3. The occurrence of this fault indicates a violated precondition resulted from the preceding operation setGreen(); this violated precondition could cause the related succeeding operation setRed() in the expected operation execution sequence NOT to be executed correctly, i.e. the traffic light device’s operation setRed() cannot be executed as expected or its execution fails.

Hence, Primary Fault 1.1 FAULT_TL_GREEN could indirectly result in the occurrence of the major fault/failure scenario of the CPS safety rule as described in Section 9.3.4.1. The second fault diagnostic solution is able to diagnose this indirectly-related primary fault. In the same manner, following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is related to the traffic light device’s operation setGreen() can be corrected and removed.

(3) Primary Fault 2.2 FAULT_IN_PC_CLEARED (as described in Table 9.3)

For diagnosing an indirectly-related primary fault, the third fault diagnostic solution we developed with the CPS TUC1 test design uses test group 2.5 TG to exercise test operation 2.5 TO `clear`(), which is verified by its associated (postcondition) test contract 2.5 ETC `checkState( inPhotoCell`, "IN_PC_CLEARED") and test state "IN_PC_CLEARED" in the CPS TUC1 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed a fault: the fault is related to the in-PhotoCell sensor device operated at the PAL entry point, where this CPS device fails in the execution of operation `clear`(), causing the in-PhotoCell sensor device NOT to be in the correct control state of "IN_PC_CLEARED" as expected. This is Primary Fault 2.2 FAULT_IN_PC_CLEARED as described in Table 9.3. The occurrence of this fault indicates that the current car might have not finished its access to the PAL entry point. Consequently, this fault could lead to a violated precondition resulting from the preceding operation `clear`(); this violated precondition could cause the related succeeding operation `setRed`() in the expected operation execution sequence NOT to be executed correctly, i.e. the traffic light device's operation `setRed`() cannot be executed as expected or its execution fails.

Thus, Primary Fault 2.2 FAULT_IN_PC_CLEARED could indirectly result in the occurrence of the major fault/failure scenario of the CPS safety rule as described in Section 9.3.4.1. The third fault diagnostic solution is able to diagnose this indirectly-related primary fault. In the same way, following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related operation `clear`() of the in-PhotoCell sensor device.

(4) Combined faults of the above three individual CPS primary faults

To diagnose the combined faults related to the traffic light device and the in-PhotoCell sensor device, the fault diagnostic solution needs to combine the above three individual fault diagnostic solutions. Based on the above (1) to (3), the combined diagnostic solution can detect and diagnose the possible combinations of these three CPS primary faults, and the combined faults can be corrected and removed in the following fault-related operations:

(a) the traffic light device's operation `setRed`(), and/or

(b) the traffic light device's operation `setGreen`(), and/or

(c) the in-PhotoCell sensor device's operation `clear`().

9.3.5.3 Evaluation Summary: Adequate Component Fault Coverage and Diagnostic Solutions and Results

Based on the three evaluation examples and relevant discussions for the MBSCT evaluation with the CPS case study (especially in Section 9.3.4, Section 9.3.5.1, Table 9.3 and Section 9.3.5.2; Section B.7 and Section B.8 in Appendix B), the evaluation of adequate component fault coverage and diagnostic solutions can be summarised as shown in Table 9.4. This table shows three main evaluation result sets (in the first three rows) that are assessed in terms of the number of different test scenarios, directly-related primary faults, indirectly-related primary faults and fault diagnostic solutions for the three CPS special testing requirements.

Table 9.4 Evaluation Summary: Adequate Component Fault Coverage and Diagnostic Solutions and Results (CPS Case Study)

	Special Testing Requirement	Test Scenario	No. of Directly-Related Faults	No. of Indirectly-Related Faults	No. of Directly/ Indirectly Related Faults	No. of Fault Diagnostic Solutions	Adequate Component Fault Coverage	Adequate Fault Diagnostic Solutions	Testing Requirement Fulfilment
	#1: Parking Access Safety Rule	CPS TUC1	1	3	4	4	Yes	Yes	Yes
	#2: Parking Pay-Service Rule	CPS TUC2	1	1	2	2	Yes	Yes	Yes
	#3: Parking Service Security Rule	CPS TUC3	1	3	4	4	Yes	Yes	Yes
Total	3	3	3	7	10	10	Yes	Yes	Yes

These evaluation result sets have drawn the following conclusions:

(1) Based on the relevant FDD evaluation (as described in Section 9.3.4.1, Section 9.3.5.1, Table 9.3 and Section 9.3.5.2.1 above), the first evaluation result set (in Table 9.4) concludes that the CPS TUC1 test design can employ the four (4) fault diagnostic solutions we developed to adequately cover and diagnose the combined faults of four (4) directly/indirectly-related primary faults. Accordingly, this achieves adequate component fault coverage and adequate fault diagnostic solutions, and fulfils the first CPS special testing requirement #1: Parking Access Safety Rule.

(2) Based on the relevant FDD evaluation (in Section 9.3.5.1 and Table 9.3 above; Section B.7.1 and Section B.8.1 in Appendix B), the second evaluation result set (in Table 9.4)

concludes that the CPS TUC2 test design can employ the two (2) fault diagnostic solutions we have developed to adequately cover and diagnose the combined faults of two (2) directly/indirectly-related primary faults. Accordingly, this achieves adequate component fault coverage and adequate fault diagnostic solutions, and fulfils the second CPS special testing requirement #2: Parking Pay-Service Rule.

(3) Based on the relevant FDD evaluation (in Section 9.3.5.1 and Table 9.3 above; Section B.7.2 and Section B.8.2 in Appendix B), the third evaluation result set (in Table 9.4) concludes that the CPS TUC3 test design can employ the four (4) fault diagnostic solutions we developed to adequately cover and diagnose the combined faults of four (4) directly/indirectly-related primary faults. Accordingly, this achieves adequate component fault coverage and adequate fault diagnostic solutions, and fulfils the third CPS special testing requirement #3: Parking Service Security Rule.

(4) Finally (in the last row in Table 9.4), our FDD evaluation concludes that the CPS test design can employ the ten (10) fault diagnostic solutions developed in the three (3) core test scenarios to adequately cover and diagnose the combined faults of ten (10) directly/indirectly-related primary faults to fulfil all the three (3) CPS special testing requirements. As the result of FDD evaluation, we conclude that the effectiveness of the MBSCT testing capability #6 (for adequate component fault coverage and diagnostic solutions) can be achieved as required.

9.4 Case Study: Automated Teller Machine System

The testing of the Automated Teller Machine (ATM) system is the second major case study undertaken in this research, with the purpose of further validating and evaluating the core characteristic testing capabilities of the MBSCT methodology and its framework (as described in Section 9.2). This section reports important testing aspects and evaluation results of the ATM case study in terms of adequate test artefact coverage (in Section 9.4.2), component testability improvement (in Section 9.4.3), and FDD evaluation (for fault case scenario diagnosis and diagnostic solution design, adequate component fault coverage, fault diagnostic solutions and results) (in Section 9.4.4). Other relevant testing aspects (such as test model construction, model-based component test development, etc.) and evaluation results are included in Appendix C. The ATM system in our case study is described much more comprehensively and rigorously than a prototype in [124] [78]. The full ATM case study has been described earlier in [178].

9.4.1 Special Testing Requirements

An overview of the ATM system is described in Appendix C, including the main ATM operations and requirements, and core ATM transactions. This section describes the main special testing requirements to assure high quality ATM-based banking services. In particular, we have identified a set of special quality requirements for supporting secure and reliable banking services for the core ATM transactions in the ATM system. Accordingly, these special quality requirements become the central focus of testing and evaluation undertaken in the ATM case study.

Section C.2 in Appendix C describes a set of eight important ATM special testing requirements we have identified particularly with regard to the first two core ATM transactions "Inquire Balance" and "Withdraw Cash" in the ATM system. By demonstrating a series of three illustrative test evaluation examples (#1, #2, and #3) selected from the ATM case study, we specifically aim to validate and evaluate how the core MBSCT testing capabilities can be effectively applied to test the ATM system to fulfil the three most important ATM special testing requirements (#3, #7 and #8). For the evaluation example #3 shown in this chapter, we describe the selected ATM special testing requirement #8 in Section 9.4.1, and relevant testing aspects and evaluation results about this special testing requirement specifically in subsequent Sections 9.4.2 to 9.4.4. Appendix C presents relevant testing aspects and evaluation results in association with the other two ATM special testing requirements #3 and #7 using the evaluation examples #1 and #2 (especially in Sections C.6 to C.8).

The selected "Special Testing Requirement #8: Account Balance Validation" is specified as follows:

(1) Special Testing Requirement #8: Account Balance Validation – validating the available credit balance of the customer-selected account that can be transacted correctly in the ATM system

In the ATM system, the customer-selected account must have a sufficient credit balance available for correctly performing certain ATM transactions, such as "Withdraw Cash" or "Transfer Money". Account balance validation has the following specific requirements:

(a) The customer-selected account must have previously been validated correctly as described in the above "Special Testing Requirement #7: Account Selection Validation".

(b) The available credit balance of the customer-selected account must be sufficient, and must be greater than or equal to the transaction amount (i.e. the customer-requested amount of money that can be transacted correctly in the customer-selected ATM transaction).

9.4.2 Evaluating Test Artefact Coverage and Adequacy

This section evaluates test artefact coverage and adequacy for testing the ATM system, which is based on the test models and component test design undertaken for the ATM case study (as described in Section C.4 to Section C.5 in Appendix C). Adequate test artefact coverage can be assessed in terms of sufficiently-covered test scenarios/sequences, sub test scenarios/sequences, test groups, test operations, test contracts and test states for the CIT purpose.

In the ATM case study, the evaluation of test artefact coverage and adequacy can be measured as shown in Table 9.5. With regard to the measurement of the number of different types of test artefacts used for the ATM component test design, there are a total of three (3) main test scenarios/sequences, ten (10) sub test scenarios/sequences, thirty-one (31) test groups, thirty-three (33) test operations, twenty-nine (29) test contracts, and twenty-nine (29) test states.

Among the total of twenty-nine (29) test states used in the ATM component test design, there are twenty-one (21) different test states used in the ATM Session, ATM TUC1 and ATM TUC2 test scenarios, but the other eight (8) test states are repeatedly used for examining different ATM transactions in the ATM TUC1 and ATM TUC2 test scenarios. In addition, as indicated in Section C.5.2 in Appendix C, there are three (3) other special test states being repeatedly used as the overall preconditions/postconditions of the test scenarios of the ATM Session, ATM TUC1 and ATM TUC2.

In the ATM case study, we employ these sufficiently-covered test artefacts with the component test design to test the ATM system. Adequate test artefact coverage can effectively aid in improving component testability, which is further evaluated in Section 9.4.3 below.

Table 9.5 Measurement of Test Artefact Coverage (ATM Case Study)

	Test Scenario	No. of Test Sequences	No. of Test Groups	No. of Test Operations	No. of Test Contracts	No. of Test States
	ATM Session	4	8	9	7	7 + 1 (8)
	ATM TUC1	3	9	10	8	8 + 1 (9)
	ATM TUC2	3	14	14	14	14 + 1 (15)
Total	3	10	31	33	29	29 + 3 (32)

9.4.3 Evaluating Component Testability Improvement

Among the five main MBSCT methodological components, the TbC technique and the TCR strategy effectively contribute to model-based component testability improvement. Chapter 4 to

Chapter 7 have previously described how to apply the MBSCT methodological components to achieve component testability improvement, especially bridging the identified "test gaps" (including both *Test-Gap #1* and *Test-Gap #2*, as described in Section 5.2.4.2).

Based on the ATM component test design (as described in Section C.5 in Appendix C), adequate test artefact coverage (as described in Section 9.4.2 above) establishes the foundation for achieving good component testability improvement. This section conducts further analysis and evaluation to show adequate test artefact coverage and component testability improvement for the CIT purpose, with regard to the effectiveness of the MBSCT testing capabilities #4 and #5. By showing the three relevant evaluation examples selected from the ATM case study, we discuss how the ATM component test design and adequate test artefact coverage can bridge the identified "test gaps" to improve component testability and to fulfil the three most important ATM special testing requirements. As indicated in Section 9.4.1, the next Section 9.4.3.1 illustrates "Evaluation Example #3: Account Balance Validation" for the ATM special testing requirements #8. Section C.6 in Appendix C describes two other evaluation examples #1 and #2 for the two ATM special testing requirements #3 and #7. Section 9.4.3.2 then provides an evaluation summary for the three evaluation examples.

9.4.3.1 Evaluation Example #3: Account Balance Validation

The ATM special testing requirement #8 (Account Balance Validation) is important in a certain test scenario of a relevant ATM TUC, e.g. the ATM TUC2 core test scenario. Account balance validation requires adequate test artefact coverage and testability for validating the available credit balance of the customer-selected account that can be transacted correctly in the ATM system. Specifically, the available credit balance of the customer-selected account (e.g. "Savings" account linked to the ATM card) must be sufficient and must be greater than or equal to the transaction amount, so that the customer-requested amount can be transacted correctly in the customer-selected ATM transaction.

Based on Section C.5 in Appendix C and Section 9.4.2 above, the component test design for the ATM TUC2 core test scenario creates a special sub test sequence #2 that can exercise and examine all three testing-required control operations of account balance, including 2.4 TO, 2.5 TO and 2.6 TO. These test operations are adequate and can bridge *Test-Gap #1*. Furthermore, the special sub test sequence #2 comprises a set of appropriately-designed test contracts, including 2.4 ETC, 2.5 ETC and 2.6 ETC. These testing-support artefacts can adequately verify each of the three testing-required control operations for account balance validation, which can bridge *Test-Gap #2*. Adequate testing artefact coverage improves component testability, enabling testing to evaluate the relevant test results of account balance validation. Therefore, the ATM component test design can improve component testability and accomplish the ATM spe-

cial testing requirement #8: Account Balance Validation.

9.4.3.2 Evaluation Summary: Adequate Test Artefact Coverage and Component Testability Improvement

Based on the three evaluation examples and relevant discussions for the MBSCT evaluation (as described in Section 9.4.2 and Section 9.4.3.1 above, and Section C.6 in Appendix C), the evaluation of adequate test artefact coverage and component testability improvement with the ATM case study can be summarised as shown as in Table 9.6. This table shows three main evaluation result sets (in three rows) that are assessed in terms of test scenarios, adequate test artefact coverage, testability improvement (i.e. bridging the "test gaps", including both *Test-Gap #1* and *Test-Gap #2*), and testing requirement fulfilment.

Table 9.6 Evaluation Summary: Adequate Test Artefact Coverage and Component Testability Improvement (ATM Case Study)

Special Testing Requirement	Test Scenario	Adequate Test Artefact Coverage	Testability Improvement		Testing Requirement Fulfilment
			Bridging Test-Gap #1	Bridging Test-Gap #2	
#3: Customer Validation	ATM Session	Yes	Yes	Yes	Yes
#7: Account Selection Validation	ATM TUC1	Yes	Yes	Yes	Yes
#8: Account Balance Validation	ATM TUC2	Yes	Yes	Yes	Yes

Our evaluation has concluded the following important points:

(1) Based on the relevant evaluation as described in Section 9.4.2 above and Section C.6.1 in Appendix C, the first evaluation result set has drawn the conclusion that the ATM component test design in the ATM Session test scenario is capable of achieving adequate test artefact coverage, improving component testability and fulfilling the ATM special testing requirement #3: Customer Validation.

(2) Based on the relevant evaluation in Section 9.4.2 above and Section C.6.2 in Appendix C, the second evaluation result set has drawn the conclusion that the ATM component test design in the ATM TUC1 test scenario is capable of achieving adequate test artefact coverage, improving component testability and fulfilling the ATM special testing requirement #7: Account Selection Validation.

(3) Based on the relevant evaluation in Section 9.4.2 and Section 9.4.3.1 above, the second evaluation result set has drawn the conclusion that the ATM component test design in the ATM TUC2 test scenario is capable of achieving adequate test artefact coverage, improving component testability and fulfilling the ATM special testing requirement #8: Account Balance Validation.

(4) Finally, our evaluation concludes that the ATM component test design with the MBSCT methodology can fulfil the three most important ATM special testing requirements for effective testing of the ATM system, and the effectiveness of the MBSCT testing capabilities #4 and #5 (for adequate test artefact coverage and component testability improvement) can be achieved as required.

9.4.4 Evaluating Component Fault Detection, Diagnosis and Localisation

Among the five main MBSCT methodological components, the TbC technique (especially, the CBFDD method) effectively contributes to component fault detection, diagnosis and localisation. Chapter 7 has previously demonstrated how to apply the MBSCT methodological components to detect, diagnose and locate component faults.

Based on the ATM component test design (as described in Section C.5 in Appendix C), adequate test artefact coverage (as described in Section 9.4.2) and component testability improvement (as described in Section 9.4.3) jointly create a solid foundation to undertake component fault detection, diagnosis and localisation. This section undertakes a further examination and evaluation for component fault detection, diagnosis and localisation for the CIT purpose, with regard to the MBSCT testing capabilities #3 and #6. By demonstrating a series of three FDD evaluation examples selected from the ATM case study, we discuss how the ATM component test design can effectively detect, diagnose and locate component faults to fulfil the three most important special testing requirements in the ATM system. Our evaluation focuses on analysing fault case scenarios to design fault diagnostic solutions (in Section 9.4.4.1), evaluating adequate component fault coverage (in Section 9.4.4.2), and evaluating fault diagnostic solutions and results (in Section 9.4.4.3) for the CIT purpose.

9.4.4.1 Analysing Fault Case Scenarios to Design Fault Diagnostic Solutions

For the FDD evaluation, this section analyses the ATM integration-related faults that cause certain major ATM failure scenarios that violate the three most important ATM special testing requirements. At the same time, we design relevant fault diagnostic solutions that can detect, di-

agnose and locate possible component faults to fulfil the three most important ATM special testing requirements. In particular, we present the three relevant FDD evaluation examples selected from the ATM case study for fault case scenario analysis and fault diagnostic solution design. The next Section 9.4.4.1.1 describes "Evaluation Example #3: Account Balance Validation" for the ATM special testing requirements #8. Section C.7 in Appendix C presents two other evaluation examples #1 and #2 for the two ATM special testing requirements #3 and #7.

Each FDD evaluation example is described in the following four main parts:

(1) Fault Case Scenario and Analysis: This part analyses the major ATM failure scenario caused by the major requirement-violating fault, and describes the impact of this major fault/failure in the ATM system, which is our main FDD focus. The fault is to be detected, diagnosed and located with the ATM component test design.

(2) Fault-Related Test Scenario: This part identifies which ATM test scenario is related to the fault under diagnosis. The ATM test scenario must cover the fault case scenario.

(3) Fault-Related ATM Device (or Fault-Related Bank Operation): This part analyses which ATM device (or which Bank operation) is related to the fault under diagnosis. Some faulty operation of the ATM device is one source of the fault (e.g. the incorrect invocation or definition of the component/class operation of the ATM device). Similarly, some faulty operation of the Bank is another source of the fault under diagnosis. Note that here the "Bank" represents the Bank ATM Server, which is mainly responsible for ATM-based banking operations in the Bank system (as described in Section C.4.1 in Appendix C).

(4) Fault Diagnostic Solution: This part describes the design of a contract-based diagnostic solution to detect and diagnose the target fault for fulfilling the relevant ATM special testing requirement. Based on the ATM component test design, fault diagnostic solutions are obtained with the CBFDD method (as described earlier in Chapter 7).

As discussed earlier in Chapter 7, a major testing strategy for developing fault diagnostic solutions with the MBSCT methodology is to design and apply appropriate basic test groups as basic test cases in fault detection, diagnosis and localisation. A *basic test group* usually comprises at least a test operation and its associated test contract, which verifies the execution of the test operation to diagnose a possible fault related to the component/class operation under test. A basic test group is also applied in conjunction with some associated test states that are used as a basis for test oracle design for test verification and fault diagnosis. A *basic fault diagnostic solution* contains at least one basic test group, and the fault diagnostic solution for diagnosing the

major ATM fault/failure scenario can incorporate multiple related basic fault diagnostic solutions.

9.4.4.1.1 Evaluation Example #3: Account Balance Validation

(1) Fault Case Scenario and Analysis

For the major fault/failure scenario of Account Balance Validation: The ATM/Bank system fails to validate the available credit balance of the customer-selected account, and/or fails to reject the customer's access to the selected account while this validation is NOT fulfilled. The correct validation requires that the available credit balance of the customer-selected account must be sufficient, and must be greater than or equal to the customer-requested amount of money to be transacted in the customer-selected ATM transaction. A validation failure would allow the customer to perform transactions on the selected account that is balance-insufficient (e.g. in the "Withdraw Cash" transaction, the customer could impermissibly overdraw the selected account that has the insufficient available credit balance), which violates the ATM special testing requirement #8: Account Balance Validation.

(2) Fault-Related Test Scenario

This fault is covered by a related ATM TUC test scenario, e.g., the ATM TUC2 core test scenario.

(3) Fault-Related ATM Device (or Fault-Related Bank Operation)

This fault is related to the Customer Console (Keypad) device, the Customer, and/or the Bank.

(4) Fault Diagnostic Solution

The fault diagnosis is CIT-related in the ATM TUC2 core test scenario. The fault diagnostic solution with the ATM TUC2 test design must incorporate certain basic fault diagnostic solutions with the following one or more related test groups (as described in Section C.5.2 in Appendix C):

(a) Test group 2.4 TG comprises test operation 2.4 TO `enterMoneyAmount`() and its associated test contract 2.4 ETC `checkState`(`customerConsole`, "MONEY_AMOUNT_ENTERED") (as postcondition), and test state "MONEY_AMOUNT_ENTERED".

(b) Test group 2.5 TG comprises test operation 2.5 TO `readMoneyAmount`() and its associated test contract 2.5 ETC `checkState`(`customerConsole`, "MONEY_AMOUNT_READ") (as postcondition), and test state "MONEY_AMOUNT_READ".

(c) Test group 2.6 TG comprises test operation 2.6 TO `validateAccountBalance(` `selectedAccountType`, `enteredMoneyAmount`) and its associated test contract 2.6 ETC `checkState(` `bank`, "ACCOUNT_BALANCE_VALIDATED") (as postcondition), and test state "ACCOUNT_BALANCE_VALIDATED".

9.4.4.2 Evaluating Adequate Component Fault Coverage

Using the MBSCT methodology, adequate component fault coverage can be achieved by applying sufficient test groups to develop fault diagnostic solutions to adequately cover and diagnose possible faults. At the same time, such adequate component fault coverage can be also evaluated by sufficiently-covered test groups and associated fault diagnostic solutions that are applied to fault diagnosis. Based on the fault case scenario analysis and fault diagnostic solution design as described in Section 9.4.4.1, we further analyse and evaluate adequate component fault coverage in the ATM case study, with regard to the three most important ATM special testing requirements.

Table 9.7 describes a comprehensive analysis and evaluation of adequate component fault coverage and diagnostic solutions for the three most important ATM special testing requirements, in terms of basic fault, fault case scenario and analysis, fault-related ATM device (or the bank), fault-related test scenario, fault diagnostic solution and test group coverage. Most table items for fault diagnosis analysis and evaluation are explained in Section 9.4.4.1. This table shows that the *major requirement-violating fault* (i.e. the major fault/failure scenario as described in Section 9.4.4.1, which violates the related ATM special testing requirement) is due to the occurrence of one of the several requirement-violating basic faults (which are associated with the Boolean operation "`or`"). A *basic fault*, which could subsequently cause the major fault/failure scenario, is covered adequately by the related basic fault diagnostic solution that contains at least one basic test group to diagnose the fault related to the component/class operation under test. In many situations, a *usual (or commonly-used) fault diagnostic solution* at an intermediate level needs to incorporate one or more basic fault diagnostic solutions (consisting of one or more basic test groups) to cover and diagnose one or more correlated basic faults for the joint testing objective. A *comprehensive fault diagnostic solution* to cover and diagnose the major requirement-violating fault must combine all requirement-related basic fault diagnostic solutions consisting of all requirement-related basic test groups. Following this fault diagnosis strategy, the ATM component test design with the MBSCT methodology can develop fault diagnostic solutions to adequately cover and diagnose the major requirement-violating faults and their correlated basic faults for the purpose of effective fault diagnosis in the ATM system.

Table 9.7 Analysis and Evaluation of Adequate Component Fault Coverage and Diagnostic Solutions (ATM Case Study)

Fault	Fault Case Scenario and Analysis	ATM Device	Bank	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
Special Testing Requirement #3: Customer Validation					
3. FAULT_CUSTOMER = FAULT_CARD **or** FAULT_PIN **or** FAULT_CUSTOMER_VALIDATED				The ATM Session	
3.1 FAULT_CARD = FAULT_CARD_INSERTED **or** FAULT_CARD_READ				The ATM Session	
3.1.1 FAULT_CARD _INSERTED The Card Reader device is NOT in the correct control state of "CARD_ INSERTED" as expected.	The Card Reader device fails to eject the ATM card that is inserted incorrectly into the card slot by the customer, and/or the ATM fails to be ready for the customer to re-insert a card for a new ATM session. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the Card Reader device cannot correctly read in the card information), or that the customer could not attempt to re-insert the ATM card correctly for accessing the ATM.	The Card Reader device		The ATM Session	ATM Session Test Design: Test group 1.1 TG comprises test operation 1.1 TO `insertCard`() and its associated test contract 1.1 ETC `checkState`(`cardReader`, "CARD_INSERTED") (as postcondition), and test state "CARD_INSERTED".
3.1.2 FAULT_CARD_READ The Card Reader device is NOT in the correct control state of "CARD_READ" as expected.	The ATM fails to read in the card information (e.g. card number) encoded on the customer-inserted ATM card, and/or fails to reject the unreadable/unacceptable card being inserted in (i.e. the Card Reader device fails to eject the inserted but unreadable/unacceptable card). This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the operation of customer validation cannot be performed correctly), or that the customer could not re-attempt to use a readable/acceptable card for accessing the ATM.	The Card Reader device		The ATM Session	ATM Session Test Design: Test group 1.2 TG comprises test operation 1.2 TO `readCard`() and its associated test contract 1.2 ETC `checkState`(`cardReader`, "CARD_READ") (as postcondition), and test state "CARD_READ".

Fault	Fault Case Scenario and Analysis	ATM Device	Bank	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
3.2 FAULT_PIN = FAULT_PIN_ENTERED `or` FAULT_PIN_READ				ATM Session	
3.2.1 FAULT_PIN _ENTERED The Keypad device is NOT in the correct control state of "PIN_ENTERED" as expected.	The ATM fails to reject the customer's PIN that is entered incorrectly by the customer from the Customer Console (Keypad) device, and/or fails to allow the three entries of the customer's PIN. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the ATM cannot correctly read in the customer's PIN), or that the customer could not attempt to re-enter another PIN correctly (within the permitted three entries) for accessing the ATM.	The Keypad device		The ATM Session	ATM Session Test Design: Test group 1.3 TG comprises test operation 1.3 TO `enterPIN`() and its associated test contract 1.3 ETC `checkState`(`customerConsole`, "PIN_ENTERED") (as postcondition), and test state "PIN_ENTERED".
3.2.2 FAULT_PIN_READ The Keypad device is NOT in the correct control state of "PIN_READ" as expected.	The ATM fails to read in the customer's PIN entered from the Customer Console (Keypad) device, and/or fails to reject the entered but unreadable/unacceptable customer's PIN, and/or fails to allow the three entries of a readable/acceptable customer's PIN. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the operation of customer validation cannot be performed correctly), or that the customer could not attempt to re-enter a readable/acceptable PIN (within the permitted three entries) for accessing the ATM.	The Keypad device		The ATM Session	ATM Session Test Design: Test group 1.4 TG comprises test operation 1.4 TO `readPIN`() and its associated test contract 1.4 ETC `checkState`(`customerConsole`, "PIN_READ") (as postcondition), and test state "PIN_READ".
3.3 FAULT_CUSTOMER _VALIDATED The Bank system is NOT in the correct control state of "CUSTOMER _VALIDATED" as expected.	The ATM/Bank system fails to validate the ATM-input customer information (e.g. card number and PIN), and/or fails to reject the customer's access to the ATM while this validation is NOT fulfilled. The correct validation requires that the inserted-card number must be valid, the entered PIN must be valid, and the ATM-input customer information must be correct and identical to the customer information stored in the Bank system. A validation failure would allow the customer to access the ATM while the customer-inserted card is invalid and/or the customer-entered PIN is invalid, which violates the ATM special testing requirement #3: Customer Validation.		Bank	The ATM Session	ATM Session Test Design: Test group 1.5 TG comprises test operation 1.5 TO `validateCustomer`(`insertedCard`, `enteredPIN`) and its associated test contract 1.5 ETC `checkState`(`bank`, "CUSTOMER_VALIDATED") (as postcondition), and test state "CUSTOMER_VALIDATED".

Fault	Fault Case Scenario and Analysis	ATM Device	Bank	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
Special Testing Requirement #7: Account Selection Validation					
7. FAULT_ACCOUNT_SELECTION = FAULT_ACCOUNT_TYPE_SELECTED **or** FAULT_ACCOUNT_TYPE_READ **or** FAULT_ACCOUNT_VALIDATED				ATM TUC1	
7.1 FAULT_ACCOUNT _TYPE_SELECTED The Display/Screen device is NOT in the correct control state of "ACCOUNT_TYPE _SELECTED" as expected.	The ATM fails to reject the account type that is selected incorrectly by the customer from the Customer Console (Display/Screen) device, and/or fails to allow re-selecting another bank account. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the ATM cannot correctly read in/accept a bank account type), or that the customer could not attempt to re-select a bank account correctly for accessing the ATM.	The Display /Screen device		The ATM TUC1	ATM TUC1 Test Design: Test group 2.1 TG comprises test operation 2.1 TO `selectAccountType()` and its associated test contract 2.1 ETC `checkState( customerConsole`, "ACCOUNT_TYPE_SELECTED") (as postcondition), and test state "ACCOUNT_TYPE_SELECTED".
7.2 FAULT_ACCOUNT _TYPE_READ The Display/Screen device is NOT in the correct control state of "ACCOUNT_TYPE _READ" as expected.	The ATM fails to read in the account type selected from the Customer Console (Display/Screen) device, and/or fails to reject the selected but unreadable/unacceptable account, and/or fails to allow re-selecting a readable/acceptable account. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the operation of account selection validation cannot be performed correctly), or that the customer could not attempt to re-select a readable/acceptable account for accessing the ATM.	The Display /Screen device		The ATM TUC1	ATM TUC1 Test Design: Test group 2.2 TG comprises test operation 2.2 TO `readAccountType()` and its associated test contract 2.2 ETC `checkState( customerConsole`, "ACCOUNT_TYPE_READ") (as postcondition), and test state "ACCOUNT_TYPE_READ".
7.3 FAULT_ACCOUNT _VALIDATED The Bank system is NOT in the correct control state of "ACCOUNT_VALIDATED" as expected.	The ATM/Bank system fails to validate the customer-selected account, and/or fails to reject the customer's access to the selected account while this validation is NOT fulfilled. The correct validation requires that the customer-selected account must be valid for the customer's account in the Bank system, must be linked to the inserted ATM card, and can be accessed by the customer to perform the customer-selected ATM transaction. A validation failure would allow the customer to perform transactions on the selected account, which violates the ATM special testing requirement #7: Account Selection Validation.		Bank	The ATM TUC1	ATM TUC1 Test Design: Test group 2.3 TG comprises test operation 2.3 TO `validateAccount( insertedCard`, `enteredPIN`, `selectedAccountType`) and its associated test contract 2.3 ETC `checkState( bank`, "ACCOUNT_VALIDATED") (as postcondition), and test state "ACCOUNT_VALIDATED".

Fault	Fault Case Scenario and Analysis	ATM Device	Bank	Test Scenario	Fault Diagnostic Solution: Test Group Coverage
Special Testing Requirement #8: Account Balance Validation					
8. FAULT_ACCOUNT_BALANCE = FAULT_MONEY_AMOUNT_ENTERED **or** FAULT_MONEY_AMOUNT_READ **or** FAULT_ACCOUNT_BALANCE_VALIDATED				ATM TUC2	
8.1 FAULT_MONEY _AMOUNT_ENTERED The Keypad device is NOT in the correct control state of "MONEY_AMOUNT _ENTERED" as expected.	The ATM fails to reject the money amount that is entered incorrectly by the customer from the Customer Console (Keypad) device, and/or fails to allow re-entering another amount of money to be transacted. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the ATM cannot correctly read in the money amount), or that the customer could not attempt to re-enter a money amount correctly for accessing the ATM.	The Keypad device		The ATM TUC2	ATM TUC2 Test Design: Test group 2.4 TG comprises test operation 2.4 TO `enterMoneyAmount`() and its associated test contract 2.4 ETC `checkState( customerConsole`, "MONEY_AMOUNT_ENTERED") (as postcondition), and test state "MONEY_AMOUNT_ENTERED".
8.2 FAULT_MONEY _AMOUNT_READ The Keypad device is NOT in the correct control state of "MONEY_AMOUNT _READ" as expected.	The ATM fails to read in the money amount correctly entered from the Customer Console (Keypad) device, and/or fails to reject the entered but unreadable/unacceptable money amount, and/or fails to allow re-entering a readable/acceptable amount of money to be transacted. This fault may cause a violated precondition for the succeeding ATM operation (e.g. this fault causes that the operation of account balance validation cannot be performed correctly), or that the customer could not attempt to re-enter a readable/acceptable money amount for accessing the ATM.	The Keypad device		The ATM TUC2	ATM TUC2 Test Design: Test group 2.5 TG comprises test operation 2.5 TO `readMoneyAmount`() and its associated test contract 2.5 ETC `checkState( customerConsole`, "MONEY_AMOUNT_READ") (as postcondition), and test state "MONEY_AMOUNT_READ".
8.3 FAULT_ACCOUNT _BALANCE_VALIDATED The Bank system is NOT in the correct control state of "ACCOUNT_BALANCE _VALIDATED" as expected.	The ATM/Bank system fails to validate the available credit balance of the customer-selected account, and/or fails to reject the customer's access to the selected account while this validation is NOT fulfilled. The correct validation requires that the available credit balance of the customer-selected account must be sufficient, and must be greater than or equal to the customer-requested amount of money to be transacted in the customer-selected ATM transaction. A validation failure would allow the customer to perform transactions on the selected account that is balance-insufficient (e.g. in the "Withdraw Cash" transaction, the customer could impermissibly overdraw the selected account that has the insufficient available credit balance), which violates the ATM special testing requirement #8: Account Balance Validation.		Bank	The ATM TUC2	ATM TUC2 Test Design: Test group 2.6 TG comprises test operation 2.6 TO `validateAccountBalance`(`selectedAccountType`, `enteredMoneyAmount`) and its associated test contract 2.6 ETC `checkState( bank`, "ACCOUNT_BALANCE_VALIDATED") (as postcondition), and test state "ACCOUNT_BALANCE_VALIDATED".

9.4.4.3 Evaluating Fault Diagnostic Solutions and Results

Based on the relevant FDD assessment in Section 9.4.4.1 to Section 9.4.4.2 (including Table 9.7) above and Section C.7 in Appendix C, this section further analyses and evaluates fault diagnostic solutions and results in more detail, with regard to the MBSCT testing capability #6. Further analysing possible component faults that violate the ATM special testing requirements, we can observe that the major requirement-violating fault is due to the occurrence of one of several relevant requirement-violating basic faults (which are associated with the Boolean operation "`or`"). For example, the major requirement-violating fault FAULT_ACCOUNT_BALANCE (which violates the ATM special testing requirement #8: Account Balance Validation) is due to the occurrence of the requirement-violating basic faults FAULT_ACCOUNT_BALANCE_VALIDATED or FAULT_MONEY_AMOUNT_READ or FAULT_MONEY_AMOUNT_ENTERED, which all subsequently violate the same ATM special testing requirement #8.

As indicated in Section 9.3.5.2, we can further classify these requirement-violating basic faults into the following two main categories:

(1) *Directly-related fault*: This type of basic fault is associated with the current operation that could directly result in the major requirement-violating fault against the related ATM special testing requirement. For example, the basic fault FAULT_ACCOUNT_BALANCE_VALIDATED is the directly-related fault for the major requirement-violating fault FAULT_ACCOUNT_BALANCE, which directly violates the ATM special testing requirement #8: Account Balance Validation (Section 9.4.4.3.1 presents more detailed discussions about diagnosing this directly-related fault).

(2) *Indirectly-related fault*: This type of basic fault is associated with a related preceding operation that could result in an intermediate fault or a violated precondition, and thus indirectly cause the same major requirement-violating fault against the related ATM special testing requirement. For example, the basic fault FAULT_MONEY_AMOUNT_READ is an indirectly-related fault for the same major requirement-violating fault FAULT_ACCOUNT_BALANCE, which subsequently violates the same ATM special testing requirement #8: Account Balance Validation (Section 9.4.4.3.1 presents more detailed discussions about diagnosing these indirectly-related faults).

For the same major requirement-violating fault, usually there might be more than one indirectly-related fault, while there is one directly-related fault in the ATM case study. Effective fault diagnostic solutions must cover and diagnose all these directly/indirectly related faults

against the same ATM special testing requirement. By illustrating the three relevant FDD evaluation examples selected from the ATM case study, we conduct a comprehensive analysis and evaluation of fault diagnostic solutions and results that adequately cover and diagnose all the major requirement-violating faults and their directly/indirectly related faults against the three most important ATM special testing requirements. Following our fault diagnosis strategy as described in Section 9.3.5.2, our fault diagnosis analysis and evaluation starts with first diagnosing the directly-related fault and then diagnosing the indirectly-related faults that are associated with the same major requirement-violating fault. The description of fault diagnosis analysis and evaluation in each FDD evaluation example is similar in principal as the result of applying the same FDD method with the MBSCT methodology, but differs in certain specific technical details when diagnosing different faults. Our major objective here is to evaluate the effectiveness of the fault diagnostic solutions developed with the MBSCT methodology.

For the three relevant FDD evaluation examples selected from the ATM case study, the next Section 9.4.4.3.1 illustrates "Evaluation Example #3: Account Balance Validation" for the ATM special testing requirements #8. Section C.8 in Appendix C describes two other evaluation examples #1 and #2 for the two ATM special testing requirements #3 and #7. Then, Section 9.4.4.4 provides a FDD evaluation summary for the three evaluation examples.

9.4.4.3.1 Evaluation Example #3: Account Balance Validation

This subsection evaluates the fault diagnostic solutions and results for diagnosing the possible faults that result in the same major requirement-violating fault FAULT_ACCOUNT_BALANCE against the ATM special testing requirement #8: Account Balance Validation. As described in Section 9.4.4.1.1 and Table 9.7 above, we develop and apply the three individual basic fault diagnostic solutions in the ATM case study. Each basic fault diagnostic solution uses a basic test group to diagnose a directly/indirectly related fault in the ATM TUC2 test scenario (as illustrated in Figure 9.2).

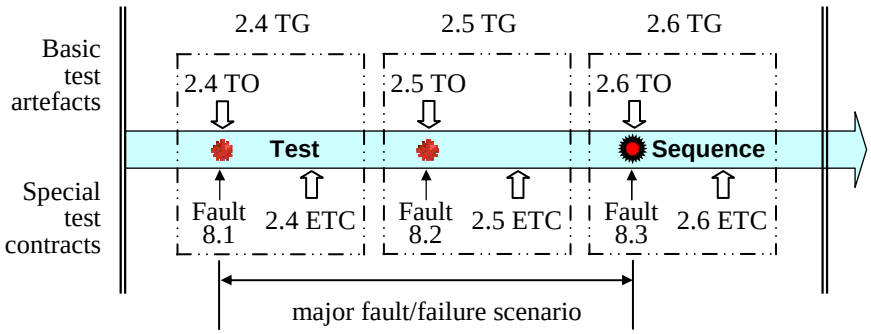


Figure 9.2 Evaluation Example #3: Account Balance Validation
(Fault Diagnostic Solutions with the ATM TUC2 Test Design)

The FDD evaluation for this major requirement-violating fault is described as follows:

(1) Basic Fault 8.3 FAULT_ACCOUNT_BALANCE_VALIDATED (as shown in Table 9.7)

To diagnose the directly-related fault in the ATM TUC2 test scenario, the ATM TUC2 test design contains the first fault diagnostic solution that uses test group 2.6 TG to exercise test operation 2.6 TO `validateAccountBalance( selectedAccountType, enteredMoneyAmount )`, which is verified by its associated test contract 2.6 ETC `checkState( bank`, "ACCOUNT_BALANCE_VALIDATED") (as postcondition) and test state "ACCOUNT_BALANCE_VALIDATED".

If the test contract returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `validateAccountBalance`() fails, causing the Bank system NOT to be in the correct control state of "ACCOUNT_BALANCE_VALIDATED" as expected. This means that the ATM/Bank system fails to validate the available credit balance of the customer-selected account, and/or the ATM fails to reject the customer's access to the selected account while this validation is NOT fulfilled. In this fault case scenario, the available credit balance of the customer-selected account is insufficient to transact the customer-requested money amount in doing a certain customer-selected ATM transaction (e.g. permitting an excess money withdrawal in the "Withdraw Cash" transaction). This accords with the basic fault 8.3 FAULT_ACCOUNT_BALANCE_VALIDATED as described in Table 9.7, and the account balance validation failure directly violates the ATM special testing requirement #8: Account Balance Validation.

Therefore, the basic fault 8.3 FAULT_ACCOUNT_BALANCE_VALIDATED is the directly-related fault that causes the major requirement-violating fault FAULT_ACCOUNT_BALANCE, which directly results in the major fault/failure scenario of Account Balance Validation as described in Section 9.4.4.1.1. The first fault diagnostic solution can diagnose this directly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related Bank's operation `validateAccountBalance`().

(2) Basic Fault 8.2 FAULT_MONEY_AMOUNT_READ (as shown in Table 9.7)

To diagnose an indirectly-related fault in the ATM TUC2 test scenario, the ATM TUC2 test design incorporates the second fault diagnostic solution that uses test group 2.5 TG to exercise test operation 2.5 TO `readMoneyAmount`(), which is verified by its associated test contract 2.5 ETC `checkState( customerConsole`, "MONEY_AMOUNT_READ") (as post-

condition) and test state "MONEY_AMOUNT_READ".

If the test contract returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the Customer Console (Keypad) device fails in the execution of operation `readMoneyAmount`(), causing the Customer Console (Keypad) device NOT to be in the correct control state of "MONEY_AMOUNT_READ" as expected. This means that the ATM fails to read in the money amount entered from the Customer Console (Keypad) device, and/or fails to reject the entered but unreadable/unacceptable money amount, and/or fails to allow the customer to re-enter a readable/acceptable amount of money to be transacted. This accords with the basic fault 8.2 FAULT_MONEY_AMOUNT_READ as described in Table 9.7. The occurrence of this fault indicates a violated precondition, causing the related succeeding operation `validateAccountBalance`() in the expected ATM TUC2 test sequence NOT to be executed correctly, i.e. this validation operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Thus, the basic fault 8.2 FAULT_MONEY_AMOUNT_READ is an indirectly-related fault that causes the directly-related fault 8.3 FAULT_ACCOUNT_BALANCE_VALIDATED, and then indirectly results in the same major requirement-violating fault FAULT_ACCOUNT_BALANCE. The second fault diagnostic solution can diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer Console device's operation `readMoneyAmount`() can be corrected and removed.

(3) Basic Fault 8.1 FAULT_MONEY_AMOUNT_ENTERED (as shown in Table 9.7)

To diagnose an indirectly-related fault in the ATM TUC2 test scenario, the ATM TUC2 test design incorporates the third fault diagnostic solution that uses test group 2.4 TG to exercise test operation 2.4 TO `enterMoneyAmount`(), which is verified by its associated test contract 2.4 ETC `checkState`(`customerConsole`, "MONEY_AMOUNT_ENTERED") (as post-condition) and test state "MONEY_AMOUNT_ENTERED".

If the test contract returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `enterMoneyAmount`() fails, causing the Customer Console (Keypad) device NOT to be in the correct control state of "MONEY_AMOUNT_ENTERED" as expected. This means that the money amount is entered incorrectly by the customer from the Customer Console (Keypad) device. While this fault occurs, the ATM fails to reject the money amount that is entered incorrectly by the customer from the Customer Console (Keypad) device, and/or fails to allow the customer to re-enter another amount of money to be transacted. This accords with the basic fault 8.1

FAULT_MONEY_AMOUNT_ENTERED as described in Table 9.7. The occurrence of this fault indicates a violated precondition, causing the succeeding operation `readMoneyAmount`() in the expected ATM TUC2 test sequence NOT to be executed correctly, i.e. this operation can not be executed as expected or its execution fails in the expected operation execution sequence.

Hence, the basic fault 8.1 FAULT_MONEY_AMOUNT_ENTERED is an indirectly-related fault that causes the indirectly-related fault 8.2 FAULT_MONEY_AMOUNT_READ, and then indirectly results in the same major requirement-violating fault FAULT_ACCOUNT_BALANCE. The third fault diagnostic solution can diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer and Customer Console device related operation `enterMoneyAmount`() can be corrected and removed.

(4) Combined faults of the above three individual directly/indirectly related faults

Based on the FDD evaluation in (1) to (3) above, a comprehensive fault diagnostic solution needs to incorporate the abovementioned three individual fault diagnostic solutions to detect and diagnose the combined faults of the above three individual directly/indirectly related faults against the same ATM special testing requirement #8: Account Balance Validation. The combined faults can be corrected and removed in the following fault-related operations:

(a) the Bank's operation `validateAccountBalance`(), and/or

(b) in the Customer Console device's operation `readMoneyAmount`(), and/or

(c) the Customer and Customer Console device related operation `enterMoneyAmount`().

9.4.4.4 Evaluation Summary: Adequate Component Fault Coverage and Diagnostic Solutions and Results

Based on the three FDD evaluation examples and relevant discussions for the MBSCT evaluation with the ATM case study (especially in Section 9.4.4.1, Section 9.4.4.2, Table 9.13 and Section 9.4.4.3 above; Section C.7 and Section C.8 in Appendix C), the evaluation of adequate component fault coverage and diagnostic solutions and results can be summarised as shown in Table 9.8. This table shows three main evaluation result sets (in the first three rows) that are assessed in terms of the number of different test scenarios, directly-related requirement-violating faults, indirectly-related requirement-violating faults and fault diagnostic solutions for the three most important ATM special testing requirements.

Table 9.8 Evaluation Summary: Adequate Component Fault Coverage and Diagnostic Solutions and Results (ATM Case Study)

	Special Testing Requirement	Test Scenario	No. of Directly-Related Faults	No. of Indirectly-Related Faults	No. of Directly/ Indirectly Related Faults	No. of Fault Diagnostic Solutions	Adequate Component Fault Coverage	Adequate Fault Diagnostic Solutions	Testing Requirement Fulfilment
	#3: Customer Validation	ATM Session	1	4	5	5	Yes	Yes	Yes
	#7: Account Selection Validation	ATM TUC1	1	2	3	3	Yes	Yes	Yes
	#8: Account Balance Validation	ATM TUC2	1	2	3	3	Yes	Yes	Yes
Total	3	3	3	8	11	11	Yes	Yes	Yes

These evaluation result sets have drawn the following conclusions:

(1) Based on the relevant FDD evaluation (as described in Section 9.4.4.1 to Section 9.4.4.3 and Table 9.7 above; Section C.7.1 and Section C.8.1 in Appendix C), the first evaluation result set (in Table 9.8) concludes that the ATM Session test design can employ the five (5) fault diagnostic solutions we have developed to adequately cover and diagnose the combined faults of five (5) directly/indirectly-related requirement-violating faults. Accordingly, this achieves adequate component fault coverage and adequate fault diagnostic solutions, and fulfils the ATM special testing requirement #3: Customer Validation.

(2) Based on the relevant FDD evaluation (in Section 9.4.4.1 to Section 9.4.4.3 and Table 9.7 above; Section C.7.2 and Section C.8.2 in Appendix C), the second evaluation result set (in Table 9.8) concludes that the ATM TUC1 test design can employ the three (3) fault diagnostic solutions we have developed to adequately cover and diagnose the combined faults of three (3) directly/indirectly-related requirement-violating faults. Accordingly, this achieves adequate component fault coverage and adequate fault diagnostic solutions, and fulfils the ATM special testing requirement #7: Account Selection Validation.

(3) Based on the relevant FDD evaluation (in Section 9.4.4.1 to Section 9.4.4.3 and Table 9.7 above), the third evaluation result set (in Table 9.8) concludes that the ATM TUC2 test design can employ the three (3) fault diagnostic solutions we have developed to adequately cover and diagnose the combined faults of three (3) directly/indirectly-related requirement-violating faults. Accordingly, this achieves adequate component fault coverage

and adequate fault diagnostic solutions, and fulfils the ATM special testing requirement #8: Account Balance Validation.

(4) Finally (in the last row in Table 9.8), our FDD evaluation concludes that the ATM test design can employ the eleven (11) fault diagnostic solutions developed in the three (3) core test scenarios to adequately cover and diagnose the combined faults of eleven (11) directly/indirectly-related requirement-violating faults to fulfil all the three (3) most important ATM special testing requirements. As the result of FDD evaluation, we can conclude that the effectiveness of the MBSCT testing capability #6 (for adequate component fault coverage and diagnostic solutions) can be achieved as required.

9.5 Evaluation Comparison and Discussions

As the principal focus of testing and evaluation, each of our case studies has validated and evaluated specifically how the MBSCT testing capabilities can be effectively applied to fulfil the most important special testing requirements. For the purpose of comparison of evaluation results, we can compare the two case studies in the following three main evaluation aspects:

(1) A comparison for the evaluation of test artefact coverage measurement (as shown in Table 9.9)

For adequate test artefact coverage, the CPS component test design employs a total of three (3) main test scenarios/sequences, eight (8) sub test scenarios/sequences, eighteen (18) test groups, twenty-three (23) test operations, eighteen (18) test contracts, and ten (10) (different) test states in the CPS case study. In the ATM case study, the ATM component test design employs a total of three (3) main test scenarios/sequences, ten (10) sub test scenarios/sequences, thirty-one (31) test groups, thirty-three (33) test operations, twenty-nine (29) test contracts, and twenty-nine (29) test states.

This evaluation comparison is shown in Table 9.9, which is a joint summary of Table 9.1 and Table 9.5. It can be observed that the ATM case study uses more test artefacts than the CPS case study does, because the ATM system is more complex than the CPS system. In particular, the ATM system has more operations under test, and thus needs more test contracts to examine these operations.

Table 9.9 Evaluation Comparison: Test Artefacts Coverage Measurement (CPS Case Study vs. ATM Case Study)

Case Study	No. of Test Scenarios	No. of Test Sequences	No. of Test Groups	No. of Test Operations	No. of Test Contracts	No. of Test States
CPS	3	8	18	23	18	10
ATM	3	10	31	33	29	29

(2) A comparison for the evaluation of adequate test artefact coverage and component testability improvement (as shown in Table 9.10)

This evaluation comparison is shown in Table 9.10, which summarises both Table 9.2 and Table 9.6. Our evaluation from the presented case studies has concluded that the effectiveness of the MBSCT testing capabilities #4 and #5 (for adequate test artefact coverage and component testability improvement) can be achieved as required by fulfilling the most important special testing requirements.

Table 9.10 Evaluation Comparison: Adequate Test Artefact Coverage and Component Testability Improvement (CPS Case Study vs. ATM Case Study)

Case Study	No. of Special Testing Requirements	No. of Test Scenarios	Adequate Test Artefact Coverage	Testability Improvement		Testing Requirement Fulfilment
				Bridging Test-Gap #1	Bridging Test-Gap #2	
CPS	3	3	Yes	Yes	Yes	Yes
ATM	3	3	Yes	Yes	Yes	Yes

(3) A comparison for the evaluation of adequate component fault coverage and diagnostic solutions (as shown in Table 9.11)

In the CPS case study, the CPS test design can use the ten (10) fault diagnostic solutions developed in the three (3) core test scenarios to adequately cover and diagnose the combined faults of ten (10) directly/indirectly-related primary faults to fulfil all the three (3) CPS special testing requirements. In the ATM case study, the ATM test design can use the eleven (11) fault diagnostic solutions developed in the three (3) core test scenarios to adequately cover and diagnose the combined faults of eleven (11) directly/indirectly-related requirement-violating faults to fulfil all the three (3) most important ATM special testing requirements.

This evaluation comparison is shown in Table 9.11, which is a joint summary of Table 9.4 and Table 9.8. As the result of FDD evaluation, our evaluation from the presented case studies has concluded that the effectiveness of the MBSCT testing capability #6 (for adequate component fault coverage and diagnostic solutions) can be achieved as required by fulfilling the

most important special testing requirements.

Table 9.11 Evaluation Comparison: Adequate Component Fault Coverage and Diagnostic Solutions and Results (CPS Case Study vs. ATM Case Study)

Case Study	No. of Special Testing Requirements	No. of Test Scenarios	No. of Directly -Related Faults	No. of Indirectly -Related Faults	No. of Directly/ Indirectly Related Faults	No. of Fault Diagnostic Solutions	Adequate Component Fault Coverage	Adequate Fault Diagnostic Solutions	Testing Requirement Fulfilment
CPS	3	3	3	7	10	10	Yes	Yes	Yes
ATM	3	3	3	8	11	11	Yes	Yes	Yes

9.6 Summary

This chapter has reported two full case studies for comprehensive validation and evaluation of the six core characteristic testing capabilities of the MBSCT methodology and its framework. While the two case studies were selected from different component-based system application areas, they were undertaken in this research to support the common goal of methodology validation and evaluation. This chapter conducted the full CPS case study that examines all the core CPS test scenarios for a more comprehensive methodology validation and evaluation. The ATM case study was an additional case study undertaken for further methodology validation and evaluation.

As the result of the comprehensive methodology validation and evaluation presented with the full case studies, we have demonstrated and validated the MBSCT testing applicability for test model construction, model-based component test design and generation, component fault detection, diagnosis and localisation in the SCT practice (including the core MBSCT testing capabilities #1 to #3). More importantly, we have examined and evaluated the MBSCT testing effectiveness for adequate component artefact coverage, component testability improvement, adequate component fault coverage and diagnostic solutions (including the core MBSCT testing capabilities #4 to #6). Our validation and evaluation have demonstrated and confirmed that the core MBSCT testing capabilities are effective to achieve the required level of component correctness and quality by fulfilling the most important special testing requirements. Therefore, our two diverse case studies have achieved the intended major objectives of the methodology validation and evaluation as described in Section 9.2.

For the MBSCT methodology developed by this research, there are some important issues concerning areas of methodology improvements, which will be further discussed in conjunction with future work in Chapter 10.

Chapter 10
Conclusions and Future Work

This chapter concludes this thesis by revisiting the original research contributions with further discussions, and exploring important open issues concerning areas for methodology improvement and research directions for future work.

10.1 Original Contributions

Section 1.3 presented an overview of the original contributions of this research. This section provides a more detailed account of the original research contributions based on the research presented in Chapter 2 to Chapter 9.

This thesis has achieved substantial and original contributions to the Software Engineering scholarly body of knowledge in the main research areas of principal interest, including software components, software component testing, model-based testing, UML-based testing, contract-based testing, scenario-based testing, mapping-based testing, and fault detection, diagnosis and localisation. The original contributions have advanced the state of the art in these research areas, and comprise two major parts: the substantial literature review to provide a firm research foundation and the comprehensive MBSCT methodology developed as the result of this research.

Each of the individual original contributions of this thesis is further discussed below:

1. The original contributions arising from the literature review for the research foundation (as described in Chapter 2 and Chapter 3)

This thesis comprehensively reviews important concepts, principles, characteristics and techniques of the abovementioned main research areas in the current literature. This literature review is substantial and comprehensive, and has achieved a number of research results and findings (including new concepts and definitions), which constitute the first major part of the original contributions of this thesis. The comprehensive literature review and research outcomes have created a solid conceptual and methodological foundation for the development of the MBSCT methodology by this research.

1.1 In the research areas of software components and software component testing

(1) A new comprehensive taxonomy of software component characteristics (in Section 2.2.2)

This new taxonomy contains twenty-two (22) software component properties in the four

(4) main classified categories at different componentisation levels, with seven (7) new component characteristics identified and added to emphasise high-level component properties. As far as we know, our proposed taxonomy is much more informative and comprehensive than the existing component characteristic classifications in the current literature (as reviewed in Section 2.2.1 and Section 2.2.2). This new taxonomy has provided a conceptual basis for the new software component definition introduced by this research (see (2) below).

(2) A new software component definition (in Section 2.2.3)

We found that there was no single formal component definition in the current literature (as reviewed in Section 2.2.1 and Section 2.2.3). Compared with other component definitions in the current literature, this new component definition was based on the abovementioned new taxonomy and comprised new added component quality properties in terms of testability and reliability, which are crucial to assure important component attributes (e.g. component functionality and reusability) in CBSE. The MBSCT methodology has integrated this new component definition to effectively improve component testability and quality.

(3) A new definition of software component testing (in Section 2.3)

We found that there was no single formal SCT definition in the current literature (as reviewed in Section 2.3). Our proposed SCT definition describes a generic testing process and the key testing tasks in six major testing phases. Based on this new SCT definition, this research analyses important SCT characteristics, test cases and specification concepts, and different testing perspectives and needs. The MBSCT methodology has incorporated this new SCT definition.

(4) A useful taxonomy of software component testing techniques (in Section 2.5)

This useful taxonomy, which was based on component development information used for component test design and generation, illustrates the relationship between the classification of testing techniques and test levels. With support from this taxonomy, this research has focused on model-based testing for the goal of component integration and system testing.

(5) A practical taxonomy of component testability improvement approaches (in Section 2.6)

This practical taxonomy was developed based on a comparative study of component testability concepts, characteristics, and improvement approaches from different stakeholder perspectives. This research emphasised component testability as a key property to support component quality and achieve component testing effectiveness. The MBSCT methodology was shown to be capable of improving component testability and quality.

1.2 In the research areas of model-based testing and UML-based testing

(1) A study of model-based tests (in Section 3.2.4)

This thesis studied model-based tests derived from test models in two main steps in terms of abstract test cases and executable test cases. Based on this study, the CTM technique developed with the MBSCT methodology has supported test transformation from abstract test cases to concrete test cases suitable for test execution (in Chapter 4 and Chapter 8).

(2) A new definition of model-based testing (in Section 3.2.5)

We found that there was no single formal MBT definition in the current literature (as reviewed in Section 3.2.1). In addition to covering the main MBT tasks and goals, our proposed MBT definition had a distinguishing characteristic that the new MBT definition addressed some of the main outstanding issues in current MBT practice (as reviewed in Section 3.2.5, Section 3.2.6 and Section 3.5), and clearly emphasised the integration of MBT and MBD activities to allow both to work together as part of the SDLC. This new MBT definition created a conceptual foundation for the model-based integrated SCT process and the TCR strategy developed in the MBSCT methodology (in Chapter 4 and Chapter 5).

(3) A new test model definition (in Section 3.2.6)

We found that there was no single formal test model definition in the current literature (as reviewed in Section 3.2.6). Being conceptually consistent with the new MBT definition as above, our proposed test model definition indicated that good test models for effective MBT must be developed from transformed and improved development models, must be reasonably simple and more abstract than the concrete implementation of the SUT, and must be adequately precise for target testing objectives. This research applied these insights as a guide to develop test models with the MBSCT methodology.

(4) A new definition of UML-based testing (in Section 3.3.1)

Our proposed UBT definition was based on the new MBT definition as mentioned above, where UBT was the major type of MBT approach used in this research. Based on this UBT definition, we have shown that the MBSCT methodology is a new UBT approach to SCT, which has benefited from the advantages of the UML standard and enables the utilisation of a consistent UML-based approach and specification for effective component development and testing.

(5) A core UML subset for SCT (in Section 3.3.2)

This core UML subset was selected to provide a primary modeling foundation for effective UML-based component development and testing process/techniques developed in the MBSCT methodology. This research has demonstrated that the core UML subset selected is adequate and effective to support UML-based SCT.

(6) A study and review of use case driven testing and scenario-based testing (in Sections 3.3.2 to 3.3.3, and Sections 3.4.2 to 3.4.5)

This thesis studied the main concepts and reviewed related work of use case driven testing and scenario-based testing in MBT/UBT practice. Based on this study, the scenario-based CIT technique developed with the MBSCT methodology has aided in deriving test scenarios and test sequences for UML-based component integration testing (in Chapter 4 to Chapter 8).

2. The principal original contributions of the MBSCT methodology (as described from Chapter 4 to Chapter 9)

As the principal original contributions of this thesis, this research has introduced a novel hybrid SCT methodology – *Model-Based Software Component Testing* (MBSCT), developed a set of five major supporting methodological components, and created the three-phase testing framework to enable the MBSCT methodology to possess six main methodological features and six core testing capabilities in SCT practice.

2.1 The five major MBSCT methodological components are as follows:

(1) Model-Based Integrated SCT Process (in Chapter 4 and Chapter 5)

Our proposed SCT process integrates software component development and testing into a unified UML-based software process as part of the SDLC, and enables using a consistent UML-based approach and specification for systematically developing test models and model-based component tests with UML. Based on the proposed MBT/UBT definition, this SCT process addressed some of the main outstanding issues in current MBT practice (as reviewed in Section 3.2.5, Section 3.2.6 and Section 3.5). As a base methodological component, the proposed SCT process provided a useful process model for the entire MBSCT methodology, thus enabling MBSCT to be model-based and process-based.

(2) Scenario-Based Component Integration Testing Technique (in Chapter 4 and Chapter 5)

Our proposed CIT technique focused testing priority on identifying and constructing appropriate test scenarios and test sequences to exercise and examine crucial deliverable component functions with the associated operational use case scenarios (e.g. behavioural instances and integration scenarios). This CIT technique enabled the MBSCT methodology to be scenario-based, which specifically supports component integration testing that bridges component unit testing and component system testing.

(3) Test by Contract (TbC) Technique (in Chapter 4 to Chapter 7)

As a primary MBSCT methodological component, our proposed TbC technique intro-

duced a new notion of a test contract as a key testing-support mechanism and associated contract-based concepts, and designed a set of six contract-based test criteria in order to improve component testability and bridge the identified "*test gaps*" in MBT/UBT (as reviewed in Section 3.2.5 and Section 3.5). Based on the proposed stepwise TbC process, this TbC technique was shown to enhance test model construction, model-based component test design and generation, which enabled the MBSCT methodology to be contract-based (in Chapter 4 to Chapter 6). In addition, by further extending this TbC technique, a new contract-based fault detection and diagnosis (CBFDD) method was developed to support effective component fault diagnosis and localisation, which established the major technical foundation for component test evaluation, thus enabling the MBSCT methodology to be FDD-based (in Chapter 7).

(4) Testing-Centric Remodeling (TCR) Strategy (in Chapter 4 and Chapter 5)

Our proposed TCR strategy provided a practical guide to assist test model construction and model-based test derivation by means of test-centric model refinement, model-based testability improvement and test-centric model optimisation. This TCR strategy worked collaboratively with the corresponding MBSCT methodological components, especially the TbC technique and the scenario-based CIT technique.

(5) Component Test Mapping (CTM) Technique (in Chapter 4 and Chapter 8)

Our proposed CTM technique is a new mapping-based test derivation approach, which focused on mapping and transforming testing-related model artefacts and associated test contracts into useful test data for generating target component test cases, thus enabling the MBSCT methodology to be mapping-based.

2.2 The MBSCT framework has three main phases for undertaking UML-based SCT:

(1) Test Model Construction (in Chapter 4 and Chapter 5)

The MBSCT framework in Phase #1 applies the first four MBSCT methodological components (as described in the list of 2.1 (1) to (4) above) to build UML-based test models, which creates a solid foundation for UML-based SCT.

(2) Component Test Design and Generation (in Chapter 4 to Chapter 8)

Based on relevant UML-based test models and test artefacts, the MBSCT framework in Phase #2 undertakes component test design to attain effective FDD. The designed model-based (abstract) test cases are then mapped and transformed into concrete test cases for generating target component test cases. Component test development is supported by all five MBSCT methodological components.

(3) Component Test Evaluation (in Chapter 7 and Chapter 9)

The MBSCT framework in Phase #3 undertakes component test evaluation in conjunction with the validation and evaluation of the core MBSCT testing capabilities (as described in the list of 2.4 (1) – (6) below).

2.3 The MBSCT methodology and its framework have six main methodological features.

Supported by the above five major methodological components, the MBSCT methodology and its framework have six main methodological features that enables SCT to be model-based, process-based, scenario-based, contract-based, FDD-based, and mapping-based in the SCT practice (as described in the list of 2.1 (1) – (5) above).

2.4 The MBSCT methodology and its framework have six core testing capabilities.

The six core MBSCT testing capabilities, which are listed below, were built on the five major methodological components and the six main methodological features of the MBSCT methodology. The first three MBSCT testing capabilities were classified into the category of testing applicability, and the remaining three were classified into the category of testing effectiveness. This thesis has undertaken a comprehensive methodology validation and evaluation (in Chapter 9), which has demonstrated and confirmed that the core MBSCT testing capabilities are effective in achieving the required level of component correctness and quality:

(1) MBSCT Capability #1: test model construction
(2) MBSCT Capability #2: model-based component test design and generation
(3) MBSCT Capability #3: component fault detection, diagnosis and localisation
(4) MBSCT Capability #4: adequate test artefact coverage
(5) MBSCT Capability #5: component testability improvement
(6) MBSCT Capability #6: adequate component fault coverage and diagnostic solutions

2.5 The MBSCT methodology has integrated the six new SCT/MBT concepts/definitions.

The MBSCT methodology has integrated the six new SCT/MBT concepts/definitions developed in this research, including a taxonomy of software component characteristics, a software component definition, a SCT definition, a MBT definition, a UBT definition, and a test model definition (as described in the lists of 1.1 and 1.2 above). The MBSCT methodology has been rigorously developed based on the conceptual foundation created by these six new SCT/MBT concepts/definitions. This demonstrates that this research has made an original contribution that achieves the seamless integration of a practical testing methodology with theoretical testing concepts and principles.

10.1.1 Methodology Comparison

This section presents a methodology comparison between the MBSCT methodology and a number of representative SCT/MBT (including UBT, UML-based SCT) approaches reported in the literature (as reviewed earlier in Chapter 2 and Chapter 3). Because these representative SCT/MBT approaches have been highly cited by many research papers/work reported in the literature, it is appropriate to select them for the comparison, although admittedly it is impractical to obtain an all-inclusive or complete list of SCT/MBT approaches for the comparison. Our comparison concludes that the MBSCT methodology is a very comprehensive UML-based SCT methodology that has been developed with effective methodological components (testing techniques and processes) and testing framework, and possesses unique methodological features and testing capabilities, which all have significant advantages over the most cited representative SCT/MBT approaches reported in the literature.

Table 10.1 shows a summary of our comparison in terms of the important methodological features and testing capabilities. In this table, for a SCT/MBT approach (as reported in the related work/paper) under comparison, the "✔" symbol denotes that the approach has (or partially has) the feature/capability in the corresponding column, and the "NA" denotes that the approach has no feature/capability in the corresponding column, thus making the comparison "not applicable". Table 10.1 clearly shows that the MBSCT methodology has more "✔" symbols than the ones having by any of the most cited representative SCT/MBT approaches. The primary objective of Table 10.1 is to demonstrate the main MBSCT advantages and differences with a group of the most cited representative SCT/MBT approaches reported in the literature.

Table 10.1 Comparison Summary: the MBSCT Methodology vs. Representative SCT/MBT Approaches

Testing Approach	Test Level	UML-Models	Testing Process	Model-Based Test Derivation	Fault Diagnosis	Test Criteria	Adequate Test Artefact Coverage	Testability Improvement	Adequate Fault Coverage	Tool-Support Test Automation	Test Evaluation	Conceptual Foundation
MBSCT	component integration testing	✓	✓	✓	✓	✓ future work	✓	✓	✓	future work	two full case studies, future work	✓
Approach by Offutt & Abdurazik [103]	system testing	✓	NA	✓	NA	✓ partial, future work	✓	NA	NA	✓ partial, future work	testing examples, future work	partial
Approach by Hartmann et al. [72]	integration testing	✓	NA	✓	NA	NA	partial	NA	NA	✓	testing examples	NA
Approach by Briand & Labiche [30]	system testing	✓	✓	✓ partial, future work	NA	NA	NA	✓ partial, future work	NA	✓ partial, future work	testing examples, future work	NA
Approach by Wu et al. [162]	component integration testing	✓	NA	NA	NA	✓	✓	NA	NA	future work	testing examples, future work	NA
Approach by Offutt et al. [105]	system testing	✓	✓	✓	partial	✓	✓	NA	NA	partial, future work	one case study, future work	NA
Approach by Nebut et al. [102]	system testing	✓	✓	✓	NA	✓	✓	NA	NA	✓	three small case studies	NA

Based on this table, we analyse and examine the comparison further in the following five main aspects:

(1) We found that there was no single SCT/MBT approach reported in the literature that is developed with all of the five major MBSCT methodological components introduced by this research.

Our comparison has shown that most reported SCT/MBT approaches are developed with only some (but not all) of the five major MBSCT methodological components (as illustrated in Table 10.1). While different testing approaches may use different techniques, these major MBSCT methodological components represent the most important testing techniques and processes that a very comprehensive SCT/MBT should have, and thus significantly contribute to the development of an effective model-based component testing approach in practice.

(2) We found that there was no single SCT/MBT approach reported in the literature having all of the six main methodological features that the MBSCT methodology possesses.

Our comparison has shown that most reported SCT/MBT approaches only have some (but not all) of the six main MBSCT methodological features (as illustrated in Table 10.1) that are supported by the five major MBSCT methodological components. While different testing approaches may have different characteristics, these main MBSCT methodological features characterise an effective model-based component testing approach, and thus significantly contribute to the development of an effective model-based component testing approach in practice.

(3) We found that there was no single SCT/MBT approach reported in the literature having all of the six core testing capabilities that the MBSCT methodology possesses.

Our comparison has shown that most reported SCT/MBT approaches only have some (but not all) of the six core MBSCT testing capabilities (as illustrated in Table 10.1) that are supported by the five major MBSCT methodological components. These core MBSCT testing capabilities have covered the most important testing functionalities that an effective model-based component testing approach should have in testing practice, and thus enable a model-based component testing approach to be much more effective to achieve the required level of component correctness and quality.

(4) There is little work reported in the literature that has validated and evaluated a SCT/MBT approach comprehensively with a series of full case studies as undertaken by this research for the assessment of the MBSCT methodology.

Our comparison has shown that most reported SCT/MBT approaches are examined only with some individual testing examples or a part of a case study in the reported work, but not

with one or more full case studies or "experiments" (as illustrated in Table 10.1). Because case study research is regarded as an effective empirical study method in software engineering, this thesis has undertaken two full case studies for the comprehensive validation and evaluation of all of the MBSCT methodological components, testing framework, methodological features and testing capabilities. This provides observable and useful evidence on the effectiveness of the MBSCT methodology in SCT/MBT practice.

(5) We found that there was no single SCT/MBT approach reported in the literature having integrated all of the six new SCT/MBT concepts/definitions introduced by this research for the rigorous development of the MBSCT methodology.

Our comparison has shown that most reported SCT/MBT approaches have incorporated less theoretical SCT/MBT concepts and principles to provide the necessary conceptual foundation to support their approaches (as illustrated in Table 10.1). There is also little work that presents its own unique testing concepts and principles as an established methodological basis for the development of a testing approach. The six new SCT/MBT concepts/definitions introduced by this research are essential SCT/MBT concepts, and create a solid conceptual foundation for the rigorous development of an effective model-based component testing approach. The MBSCT methodology has integrated them together well with its unique methodological concepts and techniques to become a comprehensive UML-based SCT methodology.

However, there are notable exceptions to the last two findings (as described in (4) and (5) above) worth mentioning. For example, for the finding described in (4) above, the work by [102] (as shown in Table 10.1 and as reviewed earlier in Section 3.4.4) presented and evaluated their approach empirically with three case studies. Two other examples of exceptions are the work by [34] (as reviewed earlier in Section 3.4.1) that conducted three experiments in controlled experiment settings, and the work by [121] that presented a full experimental case study in an industrial context. Note however that the work reported for these two examples of exceptions focused on the experiments and empirical study for evaluating existing MBT approaches that have been developed by others, and thus contained more case studies or experiments, in comparison with a research paper that concentrates on the development of a new SCT/MBT approach (plus relevant testing examples). With regard to the finding described in (5) above, three examples of exceptions are [180] [70] [21]: they all focused on a literature review with a discussion of a range of testing concepts and principles, but did not engage in the development of a new software testing approach. Note that these three papers have been also highly cited by many research papers reported in the literature in the software testing domain.

10.2 Future Work

For the MBSCT methodology introduced by this research, this section discusses several important areas of methodology improvement and research directions for future work beyond the current scope of this thesis.

1. Test Automation

One important area of methodology improvement is to develop a toolset system to support test automation of the MBSCT methodology, which would automate current manual testing. Test automation is very useful because a testing approach cannot be adopted in practice without the support of effective testing tools. Test automation also helps to effectively produce and evaluate a large number of test cases to detect more software faults. As a major part of the future development of the MBSCT methodology, our research plan on the development of a toolset system for test automation is summarized as follows:

(1) A *model reader tool* that reads and retrieves the test information from UML models.

A crucial step for automated UBT is to read, analyse and retrieve the test information from UML-based test models that are represented by UML diagrams constructed by UML modeling tools. Model-based test artefacts produced by this tool are the basis for automated test case derivation. There are two main solutions to be considered for designing a model reader tool:

(a) One design solution is, when the APIs of a UML modeling tool are accessible, to employ the provided APIs to read and retrieve the test information from UML diagrams drawn by this UML modeling tool. But this design solution may be limited to a specific UML modeling tool, which could cause some problems when UML diagrams are drawn with different UML modeling tools.

(b) Another design solution is to read, parse and retrieve the test information from the UML diagrams' XMI representation files [109], which are exported from recent advanced UML modelling tools. Because the XMI (XML Metadata Interchange) is regarded as an OMG standard for an interchange format for UML models drawn by different UML modeling tools, this design solution is UML tools independent and thus is applicable when using different UML modeling tools to build UML-based test models. This design solution needs the support of XML/XMI parsers.

(2) A *test contract generator tool* that derives verifiable test contracts for component artefacts under test, which is based on contract-based test artefacts that are designed with the TbC technique in test models and are retrieved from test models by the abovementioned model reader tool.

(3) A *test case mapper tool* that maps and transforms different levels of model-based test artefacts to produce specific test case elements, which is mainly based on the CTM technique, where an investigation of more concrete test mapping procedures and test transformation algorithms may be required.

(4) A *test specification generator tool* that compiles and integrates related test case data to generate the target CTS test case specifications.

(5) A *test specification verifier tool* that verifies test case specifications during dynamic test execution. This would be a much improved version of the TPV tool used in the previous SCL project (see Appendix A for an overview and review of the SCL project) from low-level unit testing to high-level integration/system testing.

2. OCL Expressions for Test Contracts and Test Models

Currently with the MBSCT methodology, our strategy for test contract representation and implementation is to use special test operations to handle test contracts for verifying component artefacts. The special test operations for test contracts are realised to be similar to the usual operations of components or classes, and thus are able to be executable directly with component programs to support dynamic testing. This strategy is very practical and can facilitate test automation. This is a primary reason why we currently use this strategy.

It is suggested that in a future methodology development stage, OCL expressions [160] could be used for the precise specification of test contracts and test models. Because the OCL is not intended to be a programming language for writing actions or executable code, most OCL expressions are not directly executable, just as most models are not yet executable. Therefore, it would be necessary to investigate techniques and tools to transform OCL expressions into executable forms to support dynamic testing and test automation, beyond the basic use of OCL expressions for static specification and analysis.

3. More Test Criteria

Currently with the MBSCT methodology, we have introduced a set of contract-based test criteria to support adequate test artefact coverage and testability improvement with the TbC technique, and several test mapping criteria for test case derivation with the CTM technique. Further research into additional test criteria is of interest for the purposes of effective component test development and fault diagnosis.

4. More Comprehensive Methodology Evaluations

The MBSCT methodology and its framework have been demonstrated with many illustrative testing examples, and furthermore, have been validated and evaluated comprehensively

with two full case studies. One limitation of the current methodology evaluation is that we were unable to conduct an appropriate statistical and empirical analysis, because there was limited data produced from this evaluation (as shown earlier in Section 9.3.2, Section 9.3.3.2, Section 9.3.5.3, Section 9.4.2, Section 9.4.3.2, Section 9.4.4.4 and Section 9.5). Note that the evaluation was limited to a single-object study by a single subject in an academic research context, due to the constraint of available research resources and time, as described in Section 9.2.

Thus, more comprehensive evaluations are needed to conduct empirical and comparative studies concerning the applicability and effectiveness of the MBSCT methodology with other testing approaches. More test experiments also need to be undertaken with complex component-based systems and industrial case studies.

In addition, further research into the evaluation of testing costs of the MBSCT methodology is also of interest. For example, one area of evaluation is to examine the testing costs over the number of test contracts used for test case development. Another area of evaluation is to assess the fault diagnosis costs over the number of fault diagnostic solutions used for FDD and the number of component faults revealed by the fault diagnostic solutions.

10.3 Concluding Remarks

Based on our comprehensive literature review, this thesis has identified a set of the most important challenging research problems that hamper more effective utilisation of SCT/MBT. This thesis has introduced the MBSCT methodology as our research resolution to these problems. The MBSCT methodology is a comprehensive UML-based SCT methodology that possesses five major methodological components, a three-phase testing framework, six main methodological features and six core testing capabilities. This thesis has undertaken comprehensive methodology validation and evaluation, which has demonstrated and confirmed that the MBSCT methodology is effective in achieving the required level of component correctness and quality. The methodology comparison has concluded that the MBSCT methodology has significant advantages over the most-cited representative SCT/MBT approaches reported in the literature. The significance of this research is that we have achieved substantial and original contributions to the Software Engineering scholarly body of knowledge in terms of the substantial literature review and the comprehensive MBSCT methodology in the main research areas of software components, software component testing, model-based testing, UML-based testing, contract-based testing, scenario-based testing, mapping-based testing, and fault detection, diagnosis and localisation. The methodology, techniques, processes, framework, literature reviews and associated research results presented in this thesis have altogether created a solid foundation for further research into SCT/MBT, which can help to bring closer the ultimate goal of achieving effective model-based component testing and producing trusted quality software components.

References

[1] Aynur Abdurazik and Jeff Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation," *Proc. 3rd International Conference on the Unified Modeling Language: Advancing the Standard (UML'00)*, York, UK, Oct 2000. Lecture Notes in Computer Science, vol. 1939, pp. 383–395, Springer, 2000.

[2] Aynur Abdurazik, Jeff Offutt, and Andrea Baldini, "A Controlled Experimental Evaluation of Test Cases Generated from UML Diagrams," Technical Report ISE–TR–04–03, Information and Software Engineering Department, George Mason University, USA, May 2004, 6 pages. [TR online] http://cs.gmu.edu/~tr-admin/papers/ISE-TR-04-03.pdf, 6 pages, Accessed Wed 26 Nov 2008.

[3] Aynur Abdurazik, Jeff Offutt, and Andrea Baldini, "A Comparative Evaluation of Tests Generated from Different UML Diagrams: Diagrams and Data," Technical Report ISE–TR–05–04, Information and Software Engineering Department, George Mason University, USA, April 2005, 113 pages. [TR online] http://cs.gmu.edu/~tr-admin/papers/ISE-TR-05-04.pdf, 113 pages, Accessed Mon 23 Feb 2009.

[4] Shaukat Ali, Lionel C. Briand, Muhammad Jaffar-ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem, "A State-based Approach to Integration Testing Based on UML Models," *Journal of Information and Software Technology*, vol. 49, no. 11–12, pp. 1087–1106, Nov 2007, Elsevier.

[5] Anneliese Andrews, Robert France, Studipo Ghosh, and Gerald Craig, "Test adequacy criteria for UML design models," *Journal of Software Testing, Verification and Reliability*, vol. 13, no. 2, pp. 95–127, April–June 2003, John Wiley & Sons.

[6] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell, "Fundamental Concepts of Dependability," In *Proc. 3rd IEEE Information Survivability Workshop (ISW–2000)*, Boston, MA, USA, 24–26 Oct 2000, pp. 7–12.

[7] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell, "Fundamental Concepts of Dependability," LAAS Technical Report No. 01–145, Laboratory for Analysis and Architecture of Systems, LAAS–CNRS, France, April 2001.

[8] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, January–March 2004.

[9] Andrea Baldini, Alfredo Benso, and Paolo Prinetto, "System-level functional testing from UML specifications in end-of-production industrial environments," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 326–340, Aug 2005, Springer.

[10] Aritra Bandyopadhyay and Sudipto Ghosh, "Using UML Sequence Diagrams and State Machines for Test Input Generation," student paper, *Proc. 19th IEEE International Symposium on Software Reliability Engineering (ISSRE 2008)*, Seattle, Washington, USA, 10–14 Nov 2008. IEEE Computer Society Press, 2008, pp. 309–310.

[11] Aritra Bandyopadhyay and Sudipto Ghosh, "Test Input Generation using UML Sequence and State Machines Models," Proc. *2nd IEEE International Conference on Software Testing, Verification, and Validation (ICST 2009)*, Denver, Colorado, USA, 1–4 April 2009. IEEE Computer Society Press, 2009, pp. 121–130.

[12] Franck Barbier, "COTS Component Testing through Built-In Test," book chapter, in Sami Beydeda and Volker Gruhn (Eds.), *Testing Commercial-off-the-Shelf Components and Systems*, pp. 55–70, Springer, 2005.

[13] Francesca Basanieri and Antonia Bertolino, "A Practical Approach to UML-Based Derivation of Integration Tests," *Proc. 4th Intl Software Quality Week Europe (QWE 2000)*, Brussels, Belgium, 20–24 Nov 2000.

[14] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects," In Jean-Marc J´ez´equel, Heinrich Hussmann, and Stephen Cook, editors, *Proc. 5th International Conference on The Unified Modeling Language: Model Engineering, Languages, Concepts, and Tools (UML 2002)*, Dresden, Germany, 30 Sept – 04 Oct 2002. Lecture Notes in Computer Science, vol. 2460, pages 383–397, Springer, 2002.

[15] Kent Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[16] Boris Beizer, *Software Testing Techniques*. 2nd Edition, Van Nostrand Reinhold, New York, USA, 1990.

[17] Antonia Bertolino and Andrea Polini, "WCT: A Wrapper for Component Testing," *International Workshop on Scientific Engineering for Distributed Java Applications FIDJI 2002*, Luxembourg-Kirchberg, Luxembourg, 28–29 November 2002. Lecture Notes in Computer Science, vol. 2604, pp. 165–174, Springer, 2002.

[18] Antonia Bertolino, "Software Testing Research and Practice," Invited presentation at *10th International Workshop on Abstract State Machines* (ASM 2003), Taormina, Italy, March 3–7, 2003, Lecture Notes in Computer Science, vol. 2589, pp. 1–21, Springer 2003.

[19] Antonia Bertolino and Andrea Polini, "A Framework for Component Deployment Testing," *Proc. 25th Intl Conference on Software Engineering (ICSE 2003)*, Portland, Oregon USA, 3–10 May 2003. IEEE Computer Society Press, 2003, pp. 221–231.

[20] Antonia Bertolino, Eda Marchetti, and Andrea Polini, "Integration of "Components" to Test Software Components," *Proc. Intl Workshop on Test and Analysis of Component Based Systems (TACoS 2003)* (Satellite Event of ETAPS 2003), Warsaw, Poland, 13 Apr 2003. *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 44–54, Sept 2003, Elsevier Science.

[21] Antonia Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Proc. Future of Software Engineering (FOSE 2007)*, co-located with 29th ICSE 2007, Minneapolis, Minnesota, USA, 23–25 May 2007. IEEE Computer Society Press, 2007, pp. 85–103.

[22] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," *IEEE Computer*, vol. 32, no. 7, July 1999, pp. 38–45.

[23] Robert V. Binder, "Design for testability in object-oriented systems," *Communication of ACM*, vol. 37, no. 9, pp. 87–101, Sep 1994, ACM Press.

[24] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[25] Mark Blackburn, Robert Busser, and Aaron Nauman (Software Productivity Consortium, NFP), "Why model based test automation is different and what you should know to get started," in International Conference on Practical Software Quality and Testing (PSQT/PSTT 2004 East), Washington, D. C. USA, 22 – 26 March 2004.

[26] Jonas Boberg (Erlang Training and Consulting Ltd., London, UK), "Early Fault Detection with Model-Based Testing," *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG (ERLANG 2008)*, Victoria, BC, Canada, 27–27 September 2008, pp. 9–20, ACM Press.

[27] Grady Booch, *Software Components with Ada: structures, tools, and subsystems*. 3rd Edition, Addison-Wesley, 1993.

[28] Grady Booch, James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide*. 2nd Edition, Addison-Wesley, May 2005.

[29] Borland Together, http://www.borland.com/us/products/together/, Accessed Tue 14 Apr 2009.

[30] Lionel C. Briand and Yvan Labiche, "A UML-Based Approach to System Testing," *Journal of Software and Systems Modeling*, vol. 1, no. 1, pp. 10–42, Springer, Sept 2002.

[31] Lionel C. Briand, Yvan Labiche, and H. Sun, "Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code," *Software Practice and Experience (Wiley)*, vol. 33, no. 7, pp. 637–672, 05 May 2003, John Wiley & Sons.

[32] Lionel C. Briand, Jim Cui, and Yvan Labiche, "Towards Automated Support for Deriving Test Data from UML Statecharts," *Proc. ACM/IEEE 6th Int. Conference on the Unified Modeling Language: Modeling Languages and Applications (UML'03)*, 20–24 Oct 2003, San Francisco, California, USA. Lecture Notes in Computer Science, vol. 2863, pp. 249–264, Springer, October 2003.

[33] Lionel C. Briand, and Yvan Labiche, "Empirical Studies of Software Testing Techniques: Challenges, Practical Strategies, and Future Research," *Workshop on Empirical Research in Software Testing*, co-located with IEEE/ACM 26th ICSE 2004, Edinburgh, Scotland, United Kingdom, 23–28 May 2004.

[34] Lionel C. Briand, Massimiliano Di Penta, and Yvan Labiche, "Assessing and Improving State-Based Class Testing: A Series of Experiments," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 770–793, Nov 2004.

[35] Lionel C. Briand, Jim Cui, and Yvan Labiche, "Automated support for deriving test requirements from UML statecharts," Special Issue of *Journal of Software and Systems Modeling*, vol. 4, no. 4, pp. 399–423, Nov 2005, Springer.

[36] Lionel C. Briand, Yvan Labiche, and Q. Lin, "Improving State-Based Coverage Criteria Using Data Flow Information," *Proc. 16th IEEE International Conference on Software Reliability Engineering (ISSRE 2005)*, Chicago, USA, 08–11 Nov 2005. IEEE Computer Society Press, 2005, pp. 95–104.

[37] Lionel C. Briand, "A Critical Analysis of Empirical Research in Software Testing," keynote address, *1st ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Madrid, Spain, 20–21 Sept 2007. IEEE Computer Society Press, 2007, pp. 1–8.

[38] Alan W. Brown and Kurt C. Wallnau, "The Current State of CBSE," *IEEE Software*, vol. 15, no. 5, pp. 37–46, Sept/Oct 1998.

[39] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski, "What Characterizes a (Software) Component?" *Journal of Software – Concepts and Tools*, vol. 19, no. 1, pp. 49–56, June 1998, Springer.

[40] Gary Bundell, Gareth Lee, John Morris, Kris Parker and Peng Lam, "A Software Component Verification Tool," *Proc. International Conference on Software Method and Tools (SMT'2000)*, Wollongong, NSW, Australia, 06–09 November, 2000. IEEE Computer Society Press, 2000, pp. 137–146.

[41] Alessandra Cavarra, Charles Crichton, and Jim Davies, "A method for the automatic generation of test suites from object models," *Information and Software Technology*, vol. 46, no. 5, pp. 309–314, 15 April 2004, Elsevier.

[42] Alejandra Cechich, Mario Piattini and Antonio Vallecillo (Eds.), *Component-Based Software Quality: Methods and Techniques*. Lecture Notes in Computer Science, vol. 2693, Springer-Verlag, June 2003.

[43] John Cheesman and John Daniels, *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, October 2001.

[44] Ivica Crnkovic and Magnus Larsson (Eds.), *Building Reliable Component-Based Software Systems*. Artech House Inc., 2002.

[45] Ole-Johan Dahl, Edsger Wybe Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London/New York,1972.

[46] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," *Proc. Intl Conference on Software Engineering (ICSE 1999)*, Los Angeles, CA, USA, 16–22 May 1999. ACM Press, 1999, pp. 285–294.

[47] Arilo Claudio Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme Horta Travassos, "Characterization of Model-based Software Testing Approaches," Technical Report ES–713/07, PESC-COPPE/UFRJ, Universidade Federal do Rio de Janeiro, Brazil, Aug 2007, 114 pages. [TR online] http://www.cos.ufrj.br/uploadfiles/1188491168.pdf, 114 pages, Accessed Mon 05 Jan 2009.

[48] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," *Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07)*, held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta Georgia, USA, 05 Nov 2007. ACM Press, 2007, pp. 31–36.

[49] Arilo Claudio Dias Neto and Guilherme Horta Travassos, "Supporting the Selection of Model-based Testing Approaches for Software Projects," *Proc. 3rd International Workshop on Automation of Software Test (AST 2008)*, co-located with 30th IEEE/ACM ICSE 2008, Leipzig, Germany, 10–18 May 2008. ACM Press, 2008, pp. 21–24.

[50] Arilo Claudio Dias Neto, Rajesh Subramanyan, Marlon Vieira, Guilherme Horta Travassos, and Forrest Shull, "Improving Evidence about Software Technologies: A Look at Model-Based Testing," *IEEE Software*, vol. 25, no. 3, pp. 10–13, May/June 2008.

[51] Arilo Claudio Dias Neto and Guilherme Horta Travassos, "Surveying model based testing approaches characterization attributes," *Proc. 2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2008)*, Kaiserslautern, Germany, 09–10 Oct 2008. ACM Press, 2008, pp. 324–326.

[52] Arilo Claudio Dias Neto and Guilherme Horta Travassos, "Porantim: An Approach to

Support the Combination and Selection of Model-Based Testing Techniques," *Proc. 4th International Workshop on Automation of Software Test (AST 2009)*, co-located with 31st IEEE ICSE 2009, Vancouver, BC, Canada, 18–19 May 2009.

[53] Arilo Claudio Dias Neto and Guilherme Horta Travassos, "Model-based testing approaches selection for software projects," *Information and Software Technology*, vol. 51, no. 11, pp. 1487–1504, Nov 2009, Elsevier. The special section of Third IEEE International Workshop on Automation of Software Test (AST 2008); 8th International Conference on Quality Software (QSIC 2008).

[54] Trung Dinh-Trong, Sudipo Ghosh, and Robert B. France, "A Systematic Approach to Generate Inputs to Test UML Design Models," *17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*, Raleigh, North Carolina, USA, 6–10 Nov 2006. IEEE Computer Society Press, 2006, pp. 95–104.

[55] Brian Dobing and Jeffrey Parsons, "How UML is Used," *Communications of the ACM*, vol. 49, no. 5, pp. 109–113, May 2006.

[56] Stephen H. Edwards, Murali Sitaraman, Bruce W. Weide, and Joseph Hollingsworth, "Contract-Checking Wrappers for C++ Classes," *IEEE Transactions on Software Engineering*, vol. 30, no. 11, pp. 794–810, Nov 2004.

[57] Ibranhim K. El-Far and James A. Whittaker, "Model-based Software Testing," in John J. Marciniak (Ed), *Encyclopedia of Software Engineering (2 volume set)*, 2nd Edition, Wiley, Dec 2001.

[58] Thomas Erl, *Service-Oriented Architecture: Concepts, Technology and Design*. Prentice Hall, 2005.

[59] Roy S. Freedman, "Testability of Software Components," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, June 1991.

[60] Martin Fowler, *UML Distilled*: *A Brief Guide to the Standard Object Modeling Language*. 3rd Edition, Addison-Wesley, April 2004.

[61] Falk Fraikin and Thomas Leonhardt, "SeDiTeC – Testing Based on Sequence Diagrams," *Proc. 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, Edinburgh, UK, 23–27 Sept 2002. IEEE Computer Society Press, 2002, pp. 261–266.

[62] Lars Frantzen and Jan Tretmans, "Model-Based Testing of Environmental Conformance of Components," In F. S. de Boer et al. (Eds.), *Proc. 5th International symposium on Formal Methods of Components and Objects (FMCO 2006*), Amsterdam, The Netherlands, 07–10 Nov 2006. Lecture Notes in Computer Science, vol. 4709, pp. 1–25, Springer, 2007.

[63] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[64] Jerry Gao, Eugene Y. Zhu, Simon Shim, and Lee Chang, "Monitoring software components and component-based software," *Proc. 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, Taipei, Taiwan, 25–27 Oct 2000. IEEE Computer Society Press, 2000, pp. 403–412.

[65] Jerry Gao, Kamal Gupta, Shalini Gupta, and Simon Shim, "On Building Testable Software Components," *Proceeding of 1st International Conference on COTS-Based Software Systems (ICCBSS)*, Orlando, FL, USA. 4–6 Feb 2002. Lecture Notes in Computer Science, vol. 2255, pp. 108–121, Springer.

[66] Jerry Zeyu Gao, H.-S. Jacob Tsao, and Ye Wu, *Testing and Quality Assurance for Component-Based Software*. Artech House Inc., September 2003.

[67] Jerry Gao and Ming-Chih Shih, "A Component Testability Model for Verification and Measurement," *Proc. 29th Annual Intl on Computer Software and Applications Conf (COMPSAC 2005)*, Edinburgh, Scotland, 26–28 July 2005, IEEE Computer Society Press, 2005, pp. 211–218.

[68] Studipo Ghosh, Robert France, Conrad Braganza, and Nilesh Kawane, "Test Adequacy Assessment for UML Design Model Testing," *Proc. 14th Intl Symposium on Software Reliability Engineering (ISSTA 2003)*, Denver, Colorado, USA, Nov 17–20, 2003. IEEE Computer Society Press, 2003, pp. 332–343.

[69] Hans-Gerhard Gross, *Component-Based Software Testing with UML*. Springer-Verlag, 2005.

[70] Mary Jean Harrold, "Testing: A Roadmap," *Proc. of the Conf on the Future of Software Engineering (at 22nd ICSE 2000)*, Limerick, Ireland, 04–11 June 2000, ACM Press, 2000, pp. 61–72.

[71] Alan Hartman, Mika Katara, and Sergey Olvovsky, "Choosing a Test Modeling Language: a Survey," *2nd International Haifa Verification Conference (HVC 2006)*, Haifa, Israel, 23–26 October 2006. Revised Selected Papers. Lecture Notes in Computer Science, vol. 4383, Springer, 2007, pp. 204–218.

[72] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger, "UML-Based Integration Testing," *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, Portland, Oregon, USA, 21–24 Aug 2000, pp. 60–70.

[73] Jean Hartmann, Marlon Vieira, Herb Foster, and Axel Ruder, "A UML-Based Approach to System Testing," *Innovations in Systems and Software Engineering*, vol. 1, no. 1, pp. 12–24, April 2005, Springer.

[74] George T. Heineman and William T. Councill (Eds.), *Component-Based Software Engineering: putting the pieces together*. Addison-Wesley, May 2001.

[75] Martin Höst and Per Runeson, "Checklists for Software Engineering Case Study Research," *Proc. First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Madrid, Spain, 20–21 Sept 2007. IEEE Computer Society Press, 2007, pp. 479–481.

[76] IBM Rational Software, http://www-01.ibm.com/software/rational/, Accessed Wed 13 May 2009.

[77] IEEE Std 610.12–1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE Standards Board, 28 Sept 1990.

[78] Ivar Jacobson, Grady Booch and James Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, 1999.

[79] David Janzen and Hossein Saiedian, "Test-Driven Development Concepts, Taxonomy, and Future Direction," *IEEE Computer*, vol. 38, no. 9, pp. 43–50, Sept 2005.

[80] Abu Zafer Javed, Paul Anthony Strooper, and Geoffery Norman Watson, "Automated Generation of Test Cases Using Model-Driven Architecture," *Proc. 2nd International Workshop on Automation of Software Test (AST 2007)*, co-located with 29th IEEE/ACM International Conference on Software Engineering (ICSE 2007), Minneapolis, USA, 20–26 May 2007. IEEE Computer Society Press, 2008, pp. 3.

[81] JUnit, http://www.junit.org/, Accessed Tue 14 Apr 2009.

[82] Supaporn Kansomkeat and Wanchai Rivepiboon, "Automated-generating test case using UML statechart diagrams," *Proceedings of the 2003 annual research conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through technology (SAICSIT 2003)*, 17–19 Sept 2003, pp. 296–300, SAICSIT, 2003.

[83] Supaporn Kansomkeat, Jeff Offutt, Aynur Abdurazik, and Andrea Baldini, "A Comparative Evaluation of Tests Generated from Different UML Diagrams," *Proc. 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2008)*, Phuket Thailand, Wed 06 – Fri 08 August 2008. IEEE Computer Society Press, 2008, pp. 867–872.

[84] Stuart Kent, "Model Driven Engineering," *Proc. 3rd International Conference on Integrated Formal Methods (IFM 2002)*, Turku, Finland, 15–18 May 2002. Lecture Notes in Computer Science, vol. 2335, pp. 286–298, Springer, 2002.

[85] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha, "Test Cases Generation from UML State Diagrams," *IEE Proceedings – Software*, vol. 146, no. 4, pp. 187–192, August 1999, UK.

[86] Barbara Kitchenham, Lesley Pickard and Shari Lawrence Pfleeger, "Case Studies for Method and Tool Evaluation," *IEEE Software*, vol. 12, no. 4, pp. 52–62, July 1995.

[87] Kung-Kiu Lau and Zheng Wang, "Software Component Models," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, Oct 2007.

[88] Gareth Lee, John Morris, Kris Parker, Gary A Bundell and Peng Lam, "Using Symbolic Execution to Guide Test Generation," *Journal of Software Testing, Verification and Reliability*, vol. 15, no. 1, pp. 41–61, March 2005, John Wiley & Sons.

[89] Li Bao-Lin, Li Zhi-shu, Li Qing, and Chen Yan Hong, "Test Case Automate Generation from UML Sequence Diagram and OCL Expression," *2007 International Conference on Computational Intelligence and Security (CIS 2007)*, Harbin, Heilongjiang, China, 15–19 December 2007. IEEE Computer Society Press, 2007, pp. 1048–1052.

[90] Malcolm Douglas McIlroy, "Mass Produced Software Components," In Peter Naur and Brian Randell (Eds.), *Proc. NATO Software Engineering Conference*, Garmisch, Germany, 7–11 Oct 1968. NATO Science Committee, NATO, Brussels, Belgium, pp. 88–98, Jan 1969.

[91] Bertrand Meyer, "Applying Design by Contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, Oct 1992.

[92] Bertrand Meyer, *Object-Oriented Software Construction*. 2nd Edition, Prentice Hall, 1997.

[93] Bertrand Meyer, Christine Mingins, and Heinz Schmidt, "Providing Trusted Components to the Industry," *IEEE Computer*, vol. 31, no. 5, pp. 104–105, May 1998.

[94] Bertrand Meyer, "The Grand Challenge of Trusted Components," *Proc. 25th Intl Conf on Software Engineering (ICSE 2003)*, Portland, Oregon, USA, 03–10 May 2003. IEEE Computer Society Press, 2003, pp. 660–667.

[95] MOFScript, http://www.eclipse.org/gmt/mofscript/, Accessed Tue 14 Apr 2009.

[96] John Morris, Gareth Lee, Kris Parker, Gary Bundell and Chiou Peng Lam, "Software Component Certification," *IEEE Computer*, vol. 34, no. 9, pp. 30–36, September

2001.

[97] John Morris, Chiou Peng Lam, Gareth Lee, Kris Parker, and Gary A. Bundell, "Determining Component Reliability Using a Testing Index," *Proc. Australasian Computer Science Conference (ACSC 2002)*, pp. 167–176, Melbourne, VIC, Australia, February 2002.

[98] John Morris, Peng Lam, Gary Bundell, Gareth Lee and Kris Parker, "Setting a Framework for Trusted Component Trading," In A. Cechich, M. Piattini and A. Vallecillo (Eds.), *Component-Based Software Quality: Methods and Techniques*, Lecture Notes in Computer Science, vol. 2693, pp. 128–158, Springer, June 2003.

[99] Samar Mouchawrab, Lionel C. Briand, and Yvan Labiche, "Assessing, Comparing, and Combining Statechart-based Testing and Structural Testing: An Experiment," *1st ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Madrid, Spain, 20–21 Sept 2007. IEEE Computer Society Press, 2007, pp. 41–50.

[100] Glenford J. Myers, *The Art of Software Testing*, 2nd Edition, John Wiley & Sons, 2004.

[101] Clementine Nebut, Frank Fleurey, Yves Le Traon, and Jean-Marc Jezequel, "Requirements by Contracts allow Automated System Testing," *Proc. 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*, Denver, Colorado, USA, 17–20 Nov 2003. IEEE Computer Society Press, 2003, pp. 85–96.

[102] Clementine Nebut, Frank Fleurey, Yves Le Traon, and Jean-Marc Jezequel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, March 2006.

[103] Jeff Offutt and Aynur Abdurazik, "Generating Tests From UML Specifications," *Proc. 2nd International Conference on the Unified Modeling Language: Beyond the Standard (UML'99)*, Fort Collins, CO, USA, 28–30 Oct 1999. Lecture Notes in Computer Science, vol. 1723, pp. 416–429, Springer, 1999.

[104] Jeff Offutt, Yiwei Xiong and Shaoying Liu, "Criteria for Generating Specification-based Tests," *Proc. Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, Las Vegas, NV, USA, 18–21 October 1999. IEEE Computer Society Press, 1999, pp. 119–129.

[105] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann, "Generating Test Data from State-based Specifications," *Journal of Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25–53, Jan–Mar 2003, John Wiley & Sons.

[106] Object Management Group, "Model Driven Architecture," http://www.omg.org/mda/, Accessed Fri 13 Mar 2009, Fri 25 Feb 2011.

[107] Object Management Group (OMG), "OMG Unified Modeling Language Specification," Version 1.5, OMG, March 2003. [online] http://www.omg.org/cgi-bin/doc?formal/03-03-01. Accessed: Feb 2004.

[108] Object Management Group, "The Unified Modeling Language," http://www.omg.org/uml/, http://www.uml.org/, Accessed Fri 06 Mar 2009, Fri 25 Feb 2011.

[109] Object Management Group, "XML Metadata Interchange (XMI)," [online] http://www.omg.org/technology/xml/index.htm, http://www.omg.org/spec/XMI/. Accessed: Wed 16 Mar 2011.

[110] Thomas J. Ostrand and Marc J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, June 1988.

[111] Dewayne E. Perry, Susan Elliott Sim and Steve Easterbrook, "Case Studies for Software Engineers," Tutorial F2, *28th International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, 20–28 May 2006. IEEE Computer Society Press, 2006, pp. 1045–1046.

[112] Mauro Pezze and Michal Young, *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 13 April 2007.

[113] Simon Pickin, Claude Jard, Thierry Heuillard, Jean-Marc Jézéquel, and Philippe Desfray, "A UML-Integrated Test Description Language for Component Testing," In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001*, Toronto, Canada, 01 October 2001. Lecture Notes in Informatics (LNI) vol. 7, pp. 208–223. German Informatics (GI) Society, 2001.

[114] Orest Pilskalns, Anneliese Andrews, Sudipo Ghosh, and Robert France, "Rigorous Testing by Merging Structural and Behavioral UML Representations," *6th International Conference on the Unified Modeling Language (UML 2003)*, San Francisco, CA, USA, 20–24 Oct 2003. Lecture Notes in Computer Science, vol. 2863, pp. 234–248, Springer, 2003.

[115] Orest Pilskalns, Anneliese Andrews, Andrew Knight, Sudipto Ghosh, and Robert France, "Testing UML Designs," *Information and Software Technology*, vol. 49, no. 8, pp. 892–912, Aug 2007, Elsevier.

[116] Wolfgang Prenninger and Alexander Pretschner, "Abstractions for Model-Based Testing," *Proc. the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004), Electronic Notes in Theoretical Computer Science*, vol. 116, no. 19, pp. 59–71, Jan 2005, Elsevier.

[117] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*. 7th Edition, McGraw-Hill, 2010.

[118] Alexander Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel, "Model-based testing for real – The inhouse card case study," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 2–3, pp. 140–157, Mar 2004, Springer.

[119] Alexander Pretschner, "Model-Based Testing in Practice," *Proc. Intl. Symposium of Formal Methods Europe (FM 2005)*, Newcastle, UK, 18–22 July 2005, Lecture Notes in Computer Science, vol. 3582, pp. 537–541, Springer, 2005.

[120] Alexander Pretschner and Jan Philipps, "Methodological Issues in Model-Based Testing," Chapter 10 in M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, A. Pretschner (Eds), *Model-Based Testing of Reactive Systems (Advance Lectures)*, Lecture Notes in Computer Science, vol. 3472, pp. 281–291, Springer, June 2005.

[121] Alexander Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One Evaluation of Model-Based Testing and its Automation," *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*, St. Louis, MO, USA, 15–21 May 2005. ACM Press, 2005, pp. 392–401.

[122] M. Wapas Raza, "Comparison of Class Test Integration Ordering Strategies," *IEEE*

International Conference on Emerging Technologies (ICET 2005), Islamabad, Pakistan, 17–18 Sept. 2005. IEEE Computer Society Press, 2005, pp. 440–444.

[123] David S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, Jan 1995.

[124] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[125] James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*. 2nd Edition, Addison-Wesley Object Technology Series, Addison-Wesley, 2005 (July 2004).

[126] Per Runeson and Martin Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, April 2009, Springer.

[127] Nilesh Sampat, "Components and Component-Ware Development: A Collection of Component Definitions". [online] http://www.software-components.net/components/definitions/, Accessed Thu 29 Jul 2004.

[128] Philip Samuel, Rajib Mall, and Sandeep Sahoo, "UML Sequence Diagram Based Testing Using Slicing," *IEEE INDICON 2005 Conference on Control, Communicaitons and Automation*, Chennai, India, 11–13 Dec. 2005. IEEE Computer Society Press, 2005, pp. 176–178.

[129] Philip Samuel, Rajib Mall, and A.K. Bothra, "Automatic Test Case Generation Using Unified Modeling Language (UML) State Diagrams," *IET Software*, vol. 2, no. 2, pp. 79–93, April 2008.

[130] Philip Samuel and Anju Teresa Joseph, "Test Sequence Generation from UML Sequence Diagrams," *Proc. 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2008)*, Phuket Thailand, Wed 06 – Fri 08 August 2008. IEEE Computer Society Press, 2008, pp. 879–887.

[131] Monalisa Sarma, Debasish Kundu, and Rajib Mall, "Automatic Test Case Generation from UML Sequence Diagrams," *Proc. 15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, Guwahati, India, 18–21 Dec 2007. IEEE Computer Society Press, 2007, pp. 60–67.

[132] Monalisa Sarma and Rajib Mall, "Automatic generation of test specifications for coverage of system state transitions," *Information and Software Technology*, vol. 51, no. 2, pp. 418–432, February 2009, Elsevier.

[133] Michael Scheetz, Anneliese von Mayrhauser, and Robert France, "Generating test cases from an OO model with an AI planning system," *Proc. 10th Intl Symposium on Software Reliability Eng (ISSRE 1999)*, Boca Raton, Florida, USA, 01–04 Nov 1999. IEEE Computer Society Press, 1999, pp. 250–259.

[134] Douglas C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, Feb 2006.

[135] Dehla Sokenou, "Generating Test Sequences from UML Sequence Diagrams and State Diagrams," *Model-Based Testing (MOTES 2006), Workshop* im Rahmen der 36. Jahrestagung der Gesellschaft für Informatik (GI) "Informatik 2006," Dresden, 6.10.2006, INFORMATIK 2006: Informatik für Menschen – Band 2, GI-Edition: Lecture Notes in Informatics (LNI), P–94, S. 236–240, Gesellschaft für Informatik,

2006.

[136] Ian Sommerville, *Software Engineering*. 8th Edition, Addison-Wesley, 2007.

[137] Ian Sommerville, *Software Engineering*. 9th Edition, Addison-Wesley, March 2010.

[138] "SWEBOK: Guide to the Software Engineering Body of Knowledge," 2004 Edition, IEEE. [online] http://www.swebok.org, Accessed Thu 28 Jun 2007.

[139] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*. 2nd Edition, Addison-Wesley, November 2002.

[140] Tefkat – The EMF Transformation Engine, http://tefkat.sourceforge.net/, Accessed Tue 14 Apr 2009.

[141] Jan Tretmans, "Model-Based Testing with Labelled Transition Systems," in Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.), *Formal Methods and Testing: An Outcome of the FORTEST Network*, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4949, pp. 1–38, Springer 2008.

[142] Wei-Tek Tsai, Xiaoying Bai, Ray J. Paul and Lian Yu, "Scenario-Based Functional Regression Testing," *Proc. 25th Annual Intl Computer Software and Applications Conference (COMPSAC 2001)*, Chicago, IL, USA, 8–12 Oct 2001. IEEE Computer Society Press, 2001, pp. 496–501.

[143] Wei-Tek Tsai, Yinghui Na, Ray J. Paul, F. Lu, and Akihiro Saimi, "Adaptive Scenario-Based Object-Oriented Test Frameworks for Testing Embedded Systems," *Proc. 26th Annual Intl Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, UK, 26–29 Aug 2002. IEEE Computer Society Press, 2002, pp. 321–326.

[144] Wei-Tek Tsai, Lian Yu and Akihiro Saimi, "Scenario-Based Object-Oriented Test Frameworks for Testing Distributed Systems," *Proc. 9th Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003)*, 28–30 May 2003. IEEE Computer Society Press, 2003, pp. 288–294.

[145] Wei-Tek Tsai, Ray Paul, and Lian Yu, Akihiro Saimi, and Zhibin Cao, "Scenario-Based Web Service Testing with Distributed Agents," *IEICE Transaction on Information and System*, vol. E86-D, no. 10, pp. 2130–2144, Oct 2003, IEICE Japan.

[146] Wei-Tek Tsai, Akihiro Saimi, Lian Yu and Ray Paul, "Scenario-Based Object-Oriented Test Frameworks," *Proc. 2003 Third International Conference on Quality Software (QSIC 2003)*, Dallas, Texas, USA, 6–7 Nov 2003. IEEE Computer Society Press, 2003, pp. 410–417.

[147] Wei-Tek Tsai, Ray Paul, Lian Yu and Xiao Wei, "Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems," book chapter VIII in Hongji Yang (Eds.), *Software Evolution with UML and XML*, pp. 222–262, Idea Group Publishing, London, 2005.

[148] Mark Utting, "Position Paper: Model-Based Testing," *IFIP Working Conference: The VSTTE Conference – Verified Software Theories, Tools, Experiments*, ETH, Zurich, Switzerland, 10–13 Oct 2005, 9 pages.

[149] Mark Utting, Alexander Pretschner, and Bruno Legeard, "A taxonomy of model-based testing," Technical Report 04/2006, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, April 2006. 17 pages. [TR online] http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf, Accessed Wed 20 Jun 2007.

[150] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers/Elsevier, 27 Nov 2006.

[151] Jeffrey M. Voas and Keith W. Miller, "Software Testability: The New Verification," *IEEE Software*, vol. 12, no. 3, pp. 17–28, May 1995.

[152] Jeffrey M. Voas, "How Assertions Can Increase Test Effectiveness," *IEEE Software*, vol. 14, no. 2, pp. 118–119, 122, Mar/Apr 1997.

[153] Jeffrey Voas and Lora Kassab, "Using Assertions to Make Untestable Software More Testable," *Software Quality Professional*, vol. 1, no. 4, Sep 1999.

[154] Jeffrey Voas, "Composing software component 'ilities'," *IEEE Software*, vol. 18, no. 4, pp. 16–17, July/Aug 2001.

[155] Markus Voelter, "A Taxonomy of Components," *Journal of Object Technology*, vol. 2, no. 4, pp. 119–125, July-August 2003, ETH Zurich, Chair of Software Engineering.

[156] World Wide Web Consortium (W3C), "Extensible Markup Language (XML)," [online] http://www.w3.org/xml/, http://www.w3.org/standards/xml/. Accessed: October 2008, Wed 16 Mar 2011.

[157] Y. Wang, G. King, I. Court, M. Ross and G. Staples, "On Testable Object-Oriented Programming," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, pp. 84–90, July 1997.

[158] Yingxu Wang, Graham King, and Hakan Wickburg, "A Method for Built-in Tests in Component-based Software Maintenance," *3rd European Conference on Software Maintenance and Reengineering (CSMR 1999)*, Chapel of St. Agnes, University of Amsterdam, The Netherlands. 3–5 March 1999. IEEE Computer Society Press, 1999, pp. 186–189.

[159] Yingxu Wang, Graham King, Mohamed Fayad, Dilip Patel, Ian Court, Geoff Staples and Margaret Ross, "On Built-in Test Reuse in Object-Oriented Framework Design," *ACM Computing Surveys (CSUR)*, vol. 32, no. 1es, pp. 7–12, March 2000.

[160] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd Edition, Addison-Wesley Professional, 2003.

[161] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell and Anders Wesslén, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers. Boston, MA USA, 2000.

[162] Ye Wu, Mei-Hwa Chen, and Jeff Offutt, "UML-Based Integration Testing for Component-Based Software," *Proc. 2nd Intl Conference on COTS-Based Software Systems (ICCBSS 2003)*, Ottawa, Canada, 10–12 Feb 2003. Lecture Notes in Computer Science, vol. 2580, pp. 251–260, Springer, 2003.

[163] xUnit – Unit Testing Framework, http://xunit.sourceforge.net/, Accessed Tue 14 Apr 2009.

[164] Hwei Yin and James M. Bieman, "Improving Software Testability with Assertion Insertion," *Proc. International Test Conference (ITC94)*, pp. 831–839, October 1994.

[165] Weiqun Zheng, "Software Component Testing and Certification – The Software Component Laboratory Project," Technical Report CIIPS_ISERG_TR–2006–01, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[166] Weiqun Zheng, "Towards a Standard Test Specification for Software Component Testing," Technical Report CIIPS_ISERG_TR–2006–02, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[167] Weiqun Zheng, "Model-Based Software Component Testing – An UML-Based Approach to Software Component Testing," Technical Report CIIPS_ISERG_TR–2006–03, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[168] Weiqun Zheng, "Component-Based Software Development with UML and RUP/UP – Case Study: Car Parking System," Technical Report CIIPS_ISERG_TR–2006–04, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[169] Weiqun Zheng, "Model-Based Software Component Testing: A Methodology in Practice," Technical Report CIIPS_ISERG_TR–2006–05, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[170] Weiqun Zheng, "Model-Based Software Component Testing – Case Study: Car Parking System," Technical Report CIIPS_ISERG_TR–2006–06, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2006.

[171] Weiqun Zheng and Gary Bundell, "A UML-Based Methodology for Software Component Testing," *Proc. The 2007 International Conference on Software Engineering* (*ICSE 2007*), Hong Kong, 21–23 March 2007, pp. 1177–1182.

[172] Weiqun Zheng and Gary Bundell, "Model-Based Software Component Testing: A UML-Based Approach," *Proc. 6th IEEE International Conference on Computer and Information Science* (*ICIS 2007*), Melbourne, Australia, 11–13 July 2007. IEEE Computer Society Press, 2007, pp. 891–898.

[173] Weiqun Zheng, "Applying Test by Contract to Improve Software Component Testability," Technical Report CIIPS_ISERG_TR–2007–02, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2007.

[174] Weiqun Zheng and Gary Bundell, "A Framework of UML-Based Software Component Testing," book chapter 40, in Oscar Castillo, Li Xu and Sio-Iong Ao (Eds.), *Current Trends in Intelligent Systems and Computer Engineering*, Lecture Notes in Electrical Engineering, vol. 6, pp. 575–597, Springer, May 2008.

[175] Weiqun Zheng and Gary Bundell, "Test by Contract for UML-Based Software Component Testing," *Proc. 2008 IEEE International Symposium on Computer Science and its Applications* (*CSA 2008*), Hobart, Australia, Mon 13 – Wed 15 Oct 2008. IEEE Computer Society Press, 2008, pp. 377–382.

[176] Weiqun Zheng and Gary Bundell, "Contract-Based Software Component Testing with UML Models," *International Journal of Software Engineering and Its Applications*, vol. 3, no. 1, pp. 83–102, January 2009.

[177] Weiqun Zheng, "Model-Based Approaches: Models, Modeling and Testing," Technical Report CIIPS_ISERG_TR–2009–01, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2009.

[178] Weiqun Zheng, "Model-Based Software Component Testing – Case Study: Automated Teller Machine System," Technical Report CIIPS_ISERG_TR–2010–01, Centre for Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, WA, Australia, 2010.

[179] Weiqun Zheng, Gary Bundell and Terry Woodings, "UML-Based Software Component Testing," *2010 ITEE Symposium in association with the Software Engineering Forum on Progress in Software Testing*, Perth, Australia, July 2010.

[180] Hong Zhu, Patrick A. V. Hall and John H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Survey*, vol. 29, no. 4, pp. 366–427, Dec 1997.

[181] Paul Baker, Zhen Ru Dai, Jens Grabowski, Øystein Haugen, Ina Schieferdecker, and Clay Williams, *Model-Driven Testing Using the UML Testing Profile*. Springer, 08 Nov 2007.

[182] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner (Eds.), *Model-Based Testing of Reactive Systems* (*Advance Lectures*), Lecture Notes in Computer Science, vol. 3472, Springer, June 2005.

[183] Tsun S. Chow, "Testing design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[184] Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen, and Holger Pals, "From Design to Test with UML – Applied to a Roaming Algorithm for Bluetooth Devices," *Proceedings of 16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004)*, Oxford, United Kingdom, 17–19 March 2004. Lecture Notes in Computer Science, vol. 2978, pp. 33–49, Springer, 2004.

[185] Zhen Ru Dai, "UML 2.0 Testing Profile," Chapter 17, in Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner (Eds.), *Model-Based Testing of Reactive Systems (Advance Lectures)*, Lecture Notes in Computer Science, vol. 3472, pp. 497–521, Springer, June 2005.

[186] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi, "Test Selection Based on Finite State Models," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 591–603, Jun 1991.

[187] Robert M. Hierons, "Testing from a Z Specification," *Journal of Software Testing, Verification and Reliability*, vol. 7, no. 1, pp. 19–33, March 1997, Wiley.

[188] Robert M. Hierons, Sadeghipour, S., and Singh, H, "Testing a system specified using Statecharts and Z," *Information and Software Technology*, vol. 43, no. 2, pp. 137–149, February 2001, Elsevier.

[189] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.), *Formal Methods and Testing: An Outcome of the FORTEST Network*, Revised Selected Papers. Lecture Notes in Computer Science, vol. 4949, Springer, 2008.

[190] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan, "Using Formal Specifications to Support Testing," *ACM Computing Surveys (CSUR)*, vol. 41, no. 2, pp. 1–76, February 2009.

[191] David Lee, and Mihalis Yannakakis, "Principles and Methods of Testing Finite State Machines – A Survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1126, August 1996.

[192] Edward F. Moore, “Gedanken-experiments on Sequential Machines,” In Claude E. Shannon and J. McCarthy (Eds.), *Automata Studies, Annals of Mathematical Studies*, vol. 34, pp. 129–153, Princeton University Press Princeton, N.J. USA, 1956.

[193] Object Management Group, “UML Testing Profile,” http://utp.omg.org/. Accessed Wed 16 March 2011.

[194] Object Management Group, “UML Testing Profile (UTP), Versions 1.0,” http://www.omg.org/spec/UTP/1.0/PDF, formal/05-07-07.pdf. Accessed Wed 16 March 2011.

[195] Beatriz Pérez Lamancha, Pedro Reales Mateo, Ignacio Rodríguez de Guzmán, Macario Polo Usaola, and Mario Piattini Velthius, “Automated model-based testing using the UML testing profile and QVT,” *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 2009)*, Denver, Colorado, USA, 05 Oct 2009. ACM International Conference Proceedings Series, vol. 413, ACM Press, 2009.

[196] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and Axel Rennoch, “The UML 2.0 Testing Profile and its Relation to TTCN-3,” *Proc. 15th IFIP International Conference on Testing of Communicating Systems (TestCom 2003)*, Sophia Antipolis, France, 26–28 May 2003. Lecture Notes in Computer Science, vol. 2644, pp. 79–94, Springer, 2003.

[197] Alin Stefanescu, Sebastian Wieczorek, and Marc-Florian Wendland, “Using the UML testing profile for enterprise service choreographies,” *IEEE 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, Lille, France, 01–03 Sep 2010. IEEE Computer Society Press, 2010, pp. 12–19.

[198] Jan Tretmans, “Conformance testing with labelled transition systems: implementation relations and test generation,” *Computer Networks and ISDN Systems*, vol. 29, no. 1, pp. 49–79, December 1996, Elsevier.

[199] Jan Tretmans, “Model-Based Testing and Some Steps towards Test-Based Modelling,” in Marco Bernardo and Valérie Issarny (Eds.), *Proceedings of 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, Bertinoro, Italy, 13–18 June 2011. Advanced Lectures. Lecture Notes in Computer Science, vol. 6659, pp. 297–326, Springer, 2011.

[200] Marc-Florian Wendland et al (Fraunhofer FOKUS, Berlin, Germany), “UML Testing Profile Tutorial – UTP 1.1 Review and Preview Testing,” *2011 Model-Based Testing User Conference (MBTUC 2011)*, Fraunhofer Forum, Berlin, Germany, 18–21 Oct 2011.

[201] Justyna Zander, Zhen Ru Dai, Ina Schieferdecker, and George Din, “From U2TP Models to Executable Tests with TTCN-3 – An Approach to Model Driven Testing,” *Proceedings of 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom 2005)*, Montreal, Canada, 31 May – 02 June 2005. Lecture Notes in Computer Science, vol. 3502, pp. 289–303, Springer, 2005.

Appendix A
Software Component Laboratory Project

This research was partially motivated by the previous research work conducted by the Software Component Laboratory (SCL) at the Centre of Intelligent Information Processing Systems, School of Electrical, Electronic and Computer Engineering, University of Western Australia, Australia [40] [96] [97] [98] [88]. In 1999, the SCL was established as an Australian Government funded project, and supported by a grant from Software Engineering Australia (Western Australia) Ltd through the Software Engineering Quality Centres Program of the Department of Communication, Information Technology and the Arts, Australia. The SCL project linked together expertise and collaboration at the University of Western Australia, Murdoch University and Software Engineering Australia (Western Australia) Ltd.

This appendix presents an overview of the SCL project, and reviews its main limitations and remaining issues to provide a basis for the further work as part of this research. Further details about the SCL project and a comprehensive review of it can be found in [165].

A.1 The SCL Project Overview

A principle goal of the SCL project was to establish a laboratory for building reliable component software for the new and fast-growing CBSE community. The main SCL work can be summarised as follows:

(a) An XML-based *component test specification* (CTS) for specifying and representing component test cases (called *CTS test case specifications*) [40] [96] [98]

(b) A lightweight testing tool, *test pattern verifier* (TPV) for verifying CTS test case specifications in a dynamic testing environment [40] [96] [98] [88]

(c) A *component testing index* (CTI) as the rating scheme for measuring the extent of component testing [97] [98]

(d) A prototype of the *verified software component library* (VeriLib) [40] [97] [98]

The XML-based CTS and the TPV tool are outlined in Section A.2 and Section A.3, because they are used by this research as part of the development of the new MBSCT methodology.

A.2 XML-Based Component Test Specification

The XML-based CTS integrates the XML standard and technology [156] with SCT to specify and represent CTS test case specifications, which aims to support standard test specification requirements and characteristics, such as a well-defined and well-structured specification format, portability, reusability, etc. (as described in [98] [166]). The XML-based CTS has advantages over traditional test representations [24], and well caters for the needs of both component developers and users in different contexts.

A dedicated XML DTD is defined, called CTS–DTD [98], to produce well-formed and valid XML documents for CTS test case specifications. The full DTD can be restructured and decomposed into individual XML DTDs according to different characteristic categories of related test data. Table A.1 (which is adapted from [165] [166]) shows the three sub DTDs and their corresponding test documents that can be classified and created with the XML-based CTS.

Table A.1 Component Test Specification: DTD and Test Document

CTS Document Type Definition	CTS Test Document
Component Descriptor DTD (CD–DTD)	Component Descriptor Document (CDD)
Test Specification DTD (TS–DTD)	Test Specification Document (TSD)
Test Specification DTD (TS–DTD)	Result Set Document (RSD)

The most important sub DTD is the CTS TS–DTD that contains the twelve (12) main XML elements and their attributes, as shown in Figure A.1 (which is adapted from [98] [165]). The corresponding test specification document TSD is an XML document that begins with the <TestSpecification> root element and consists of a list of test sets. A *test set* (specified by the <TestSet> element) contains multiple test groups. A *test group* (specified by the <TestGroup> element) is composed of related operations. A *test operation* (specified by the <TestOperation> element) comprises several logically ordered *atomic test operations*; each of them examines a basic class operation, such as *constructor* (specified by the <TestConstructor> element) or *method* (specified by the <TestMethod> element). These structural mechanisms organise individual tests into appropriate *test sequences* to establish the logical hierarchies of tests relevant to the component under test (CUT), which are tree-structured, well-formed, simple and easy to understand and use.

To understand these structural mechanisms and test elements, Table A.2 (which is

adapted from [166]) describes the TSD (e.g. for a test example component class heap) with its main structures, elements, tags, attributes, etc, which are defined in the TS–DTD (in Figure A.1).

```
<!-- CTS Test Specification DTD -->
<!--
<?xml version="1.0" encoding="UTF-8" standalone="no"? >
<!DOCTYPE TestSpecification SYSTEM "TestSpecification.dtd" >
-->

<!ELEMENT TestSpecification ( TestSet+ ) >
  <!ATTLIST TestSpecification Name ID #REQUIRED >
  <!ELEMENT TestSet ( Desc? ( TestGroup | TestOperation | Invariant )* ) >
    <!ATTLIST TestSet Name ID #REQUIRED >
    <!ELEMENT Decs (#PCDATA) >
    <!ELEMENT TestGroup ( Desc? (TestOperation | TestGroup | Invariant)* ) >
      <!ATTLIST TestGroup Name ID #REQUIRED >
      <!ATTLIST TestGroup TargetMethod CDATA #IMPLIED >
    <!ELEMENT Invariant ( Arg* (Result | Exception)? ) >
      <!ATTLIST Invariant DataType CDATA #REQUIRED >
      <!ATTLIST Invariant TestMethod CDATA #REQUIRED >
    <!ELEMENT TestOperation ( (TestConstructor | TestMethod | Invariant)* ) >
      <!ATTLIST TestOperation Name ID #REQUIRED >
      <!ATTLIST TestOperation Pre IDREF #IMPLIED >
      <!ATTLIST TestOperation Version CDATA #IMPLIED >
      <!ELEMENT TestConstructor ( Arg*, (Result | Exception)? ) >
        <!ATTLIST TestConstructor Name CDATA #REQUIRED >
      <!ELEMENT TestMethod ( Arg*, (Result | Exception)? ) >
        <!ATTLIST TestMethod Name CDATA #REQUIRED >
        <!ATTLIST TestMethod Target CDATA #REQUIRED >
        <!ATTLIST TestMethod Static ( Y | N ) "N" >
        <!ELEMENT Arg (#PCDATA) >
          <!ATTLIST Arg Name CDATA #IMPLIED >
          <!ATTLIST Arg Source CDATA #IMPLIED >
          <!ATTLIST Arg DataType CDATA #IMPLIED >
        <!ELEMENT Result (Exp?) >
          <!ATTLIST Result Name CDATA #IMPLIED >
          <!ATTLIST Result DataType CDATA #IMPLIED >
          <!ATTLIST Result Qualification CDATA #IMPLIED >
          <!ATTLIST Result Save ( Y | N ) "N" >
          <!ELEMENT Exp (#PCDATA) >
            <!ATTLIST Exp SpecVersion CDATA #IMPLIED >
        <!ELEMENT Exception (Exp?) >
          <!ATTLIST Exception Name CDATA #IMPLIED >
          <!ATTLIST Exception DataType CDATA #REQUIRED >
          <!ATTLIST Exception Qualification CDATA #IMPLIED >
          <!ATTLIST Exception Save ( Y | N ) "N" >
```

Figure A.1 An extract of CTS Test Specification DTD (TS–DTD)

Table A.2 Test Specification Document: structures, elements, tags, attributes

Element	Structures, Elements, Tags, Attributes, Contents	Description
Test Specification	<TestSpecification Name="heap.xml"> <TestSet>+... </TestSpecification>	The TSD has only one <TestSpecification> root element containing one or more test sets (specified by the <TestSet> element) for the CUT, e.g. heap.
TestSet (Test Set)	<TestSet Name="Heap_BasicTests"> <Desc>?... <TestGroup>*... </TestSet>	A Test Set is a collection of test groups (specified by the <TestGroup> child element) or test operations (specified by the <TestOperation> child element).
Desc (Text Description)	<Desc> A set of basic tests checking operations. </Desc>	The <Desc> element contains a text description of the element where the <Desc> element is embedded.
TestGroup (Test Group)	<TestGroup Name="Constructor check"> <Desc>?... <TestOperation>*... </TestGroup>	A Test Group groups related tests together, which typically is a sequence of logically-ordered related test operations. A Test Group may recursively include other nested test groups.
TestOperation (Test Operation)	<TestOperation Name="Heap(n)"> <TestConstrcutor>*... <TestMethod>*... </TestOperation>	A Test Operation is a sequence of calls to constructor (specified by the <TestConstructor> child element) and/or method (specified by the <TestMethod> child element) of the class under test. Calls formed into a Test Operation may perform some test scenario, such as verifying object construction, testing one of methods of an object constructed, etc.
TestConstructor (Test Constructor)	<TestConstructor Name="Heap"> <Arg Name="size" DataType="int"> 0 </Arg> <Result Name="heap" DataType="Heap"/> </TestConstructor>	A Test Constructor represents a call to construct an object. It may take arguments (specified by the <Arg> child element), and return a resultant object (specified by the <Result> child element) or throw an exception (specified by the <Exception> child element). A Test Constructor is an atomic test operation.
TestMethod (Method Call)	<TestMethod Name="size" Target="heap"> <Result Name="n" DataType="int" Save="y"> <Exp>0</Exp> </Result> </TestMethod>	A Test Method represents a call to invoke a method on an object constructed. It may take arguments, and return a result or throw an exception. The object on which the Test Method is invoked is specified by the "Target" attribute, and was already constructed by a previous constructor that stored this object and specified it by the Name attribute of the <Result> element. A Test Method is an atomic test operation.
Invariant	<Invariant DataType="..." TestMethod="..."> <Args>*... <Result>?... </Invariant>	An Invariant for a class indicates that objects of the class must always hold some properties no matter what operations are applied to the objects. It takes the form of a special method, which will always return a known result (indicating if the class is satisfied with the invariant property) when the method is called.
Arg (Argument)	<Arg Name="x" DataType="int"> 5 </Arg>	An Arg argument can be either a literal or an object already constructed by the previous constructor. The data of the <Arg> element should be read and converted to the specified data type (primitive type or class) before the constructor or method is invoked. The <Arg> element specifies some test input to the constructor or method under test.
Result	<Result Name="empty" DataType="boolean" Save="y"> <Exp>true</Exp> </Result>	A Result determines the expected data (specified by the <Exp> child element) to be returned from a method that is successfully executed. The actual returned value of the method under test is stored in the execution environment by the Name attribute, and is to be saved in the RSD accompanying the TSD when the Save="Y" attribute is present. The expected data should be read and converted to the specified data type before it is compared with the actual returned value.
Exp (Expression)	<Exp>true</Exp>	An Exp expression represents the expected data (primitive result) of the method under test.

Exception	<Exception Name="..." DataType="..." Save="Y"> <Exp>? </Exception>	An Exception indicates that the method under test is expected to throw an exception as a result rather than returning normally. The exception value is to be saved in the RSD as the specified class type of the exception if the Save="Y" attribute is present.

A.3 Test Pattern Verifier

The TPV tool enables component testers to execute and examine component tests specified by the XML-based CTS, which supports executability of component tests and *verifiability* of software components. In the testing environment, after the tester selects a test specification (an XML document) for the CUT, the TPV opens, reads, parses and stores it in an internal file structure that is similar to the XML document. The TPV applies the tests to the CUT, and checks the actual test results against:

(a) The expected test results specified by Exp elements in the test specification (especially when the selected test specification is executed the first time for testing); or

(b) The historical test results previously stored in the ResultSet documents for the CUT (especially when the same test specification is rerun for regression testing).

Figure A.2 shows the TPV's main GUI screen after a CTS test case specification has been loaded and a group of tests run (e.g. for component class heap) [98]. The main frame contains three panels:

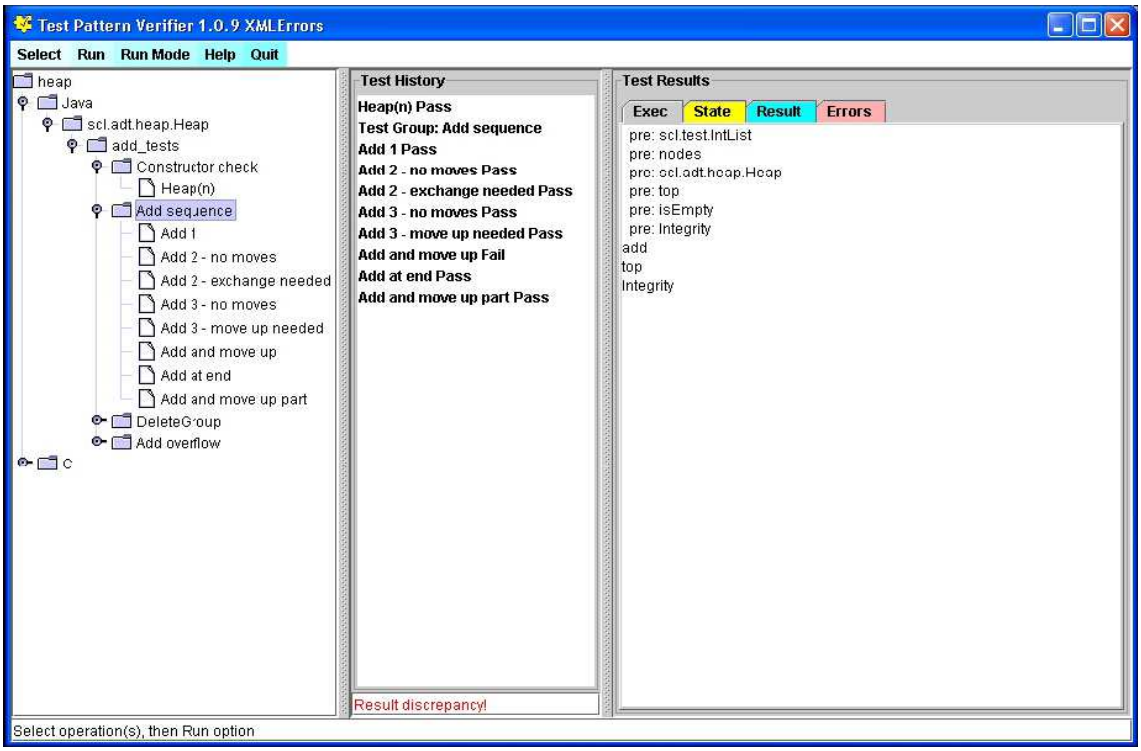


Figure A.2 Main TPV GUI: test selection, history and results panels

(1) The left-most panel displays the content of the test specification as a test set: test group: operation hierarchy. The tester can select one of the run options to run the appropriate test set, group or single operation selected.

(2) The middle panel shows a history of test executions to date combined with simple pass/fail statistics for each test execution.

(3) The right-most panel has several tabs, which enable the tester to view the execution history for individual method invocations, the state of the run time environment, error reports and results from each constructor or method call invocation.

A.4 Main Limitations and Remaining Issues

This section analyses and reviews some of the main limitations and remaining issues of the previous SCL work in order to provide a basis for the further work as part of this research, which can be summarised as follows:

(1) Component test design and generation in the previous SCL work

(a) Did not provide a systematic approach or framework on how to design and generate component tests conforming to the standard XML-based CTS.

The XML-based CTS provides a XML-styled notation for test specification and representation with a standard, well-defined and well-structured format. Although it is a novel approach towards the standardisation of test specifications, the previous SCL work did not provide a testing method for test design and generation in practice. It is necessary to undertake further research to investigate useful approaches for developing CTS test case specifications.

(b) Did not provide test criteria to assist design and evaluation of CTS test case specifications.

Using the specification notation of the XML-based CTS does not mean that any test cases represented in the CTS format are “good tests” in realising testing effectiveness. In fact, effective tests are principally measured with appropriate test criteria and requirements.

(c) Did not correlate SBT/MBT/UMBT approaches with design and generation of CTS test case specifications

The previous SCL work gave some IBT examples that derived test cases based on component programs for code-based unit testing [96] [98]. A major deficiency of the SCL work is

lacking a practical methodology for test design and generation particularly pertinent to SBT/MBT/UBT approaches for high-level testing purposes.

(2) Test levels

The previous SCL work did not well address important testing issues that can effectively support integration and system testing. Because software components are developed mainly for reuse and integration in component applications and component-based systems, component users are usually concerned much more about component integration and system testing for component software quality at higher levels.

(3) Fault detection and diagnosis

The previous SCL work did not address the important testing issue of fault detection and diagnosis, which is a crucial measurement of component quality. Fault detection and diagnosis is one of most important testing capabilities that an effective testing approach should have.

(4) Component testability and its improvement

The previous SCL work did not address the important testing issue of component testability and its improvement, which is essential to assist effective component test development to detect and diagnose possible component faults.

This research was partially motivated by the previous SCL project, with the aim to address its main limitations with regard to model-based component test design and generation, component integrations testing, component testability and its improvement, and component fault detection and diagnosis. By bridging these gaps in the previous SCL work, the new MBSCT methodology is developed to overcome these remaining problems to achieve a desirable level of SCT effectiveness.

Appendix B
Case Study: Car Parking System

The testing of the Car Parking System (CPS) is the first case study that is used throughout this thesis, in order to validate and evaluate the core characteristic testing capabilities of the MBSCT methodology and its framework. We have also used the CPS case study as a major source of illustrative examples throughout Chapter 5 to Chapter 8, and Chapter 9 has presented the most important contents of the CPS case study. This appendix provides the background and supplementary information about the CPS case study. The full CPS case study has been described earlier in [168] [170].

B.1 Overview of the CPS System

This section presents an overview of the CPS system. The CPS system is a typical access control system to provide public parking services. The CPS system employs a set of parking control devices to monitor, coordinate and regulate a flow of cars accessing the parking access lane (PAL) for parking cars in the area of parking bays. The CPS system comprises five individual parking control devices that are located in three main control points along the PAL (as illustrated in Figure B.1).

The following describes the main system operations and functional requirements for the CPS system:

(1) The first control point is the entry point, which is jointly controlled by the Traffic Light device and the In-PhotoCell Sensor device.

(a) The Traffic Light device controls a car's accessing the PAL entry point.

- The Traffic Light device displays a GREEN signal to permit the waiting car to enter the PAL;
- The Traffic Light device displays a RED signal to disallow the next car to enter the PAL and the next car must wait for access permission.

(b) The In-PhotoCell Sensor device senses whether or not the current car is accessing the PAL entry point.

- First, the In-PhotoCell Sensor device senses that the PAL entry point has been occupied by the entering car, when this car is accessing the PAL entry point;
- Then, the In-PhotoCell Sensor device senses that the PAL entry point has been cleared by the same entering car, after this car has finished accessing the PAL entry point.

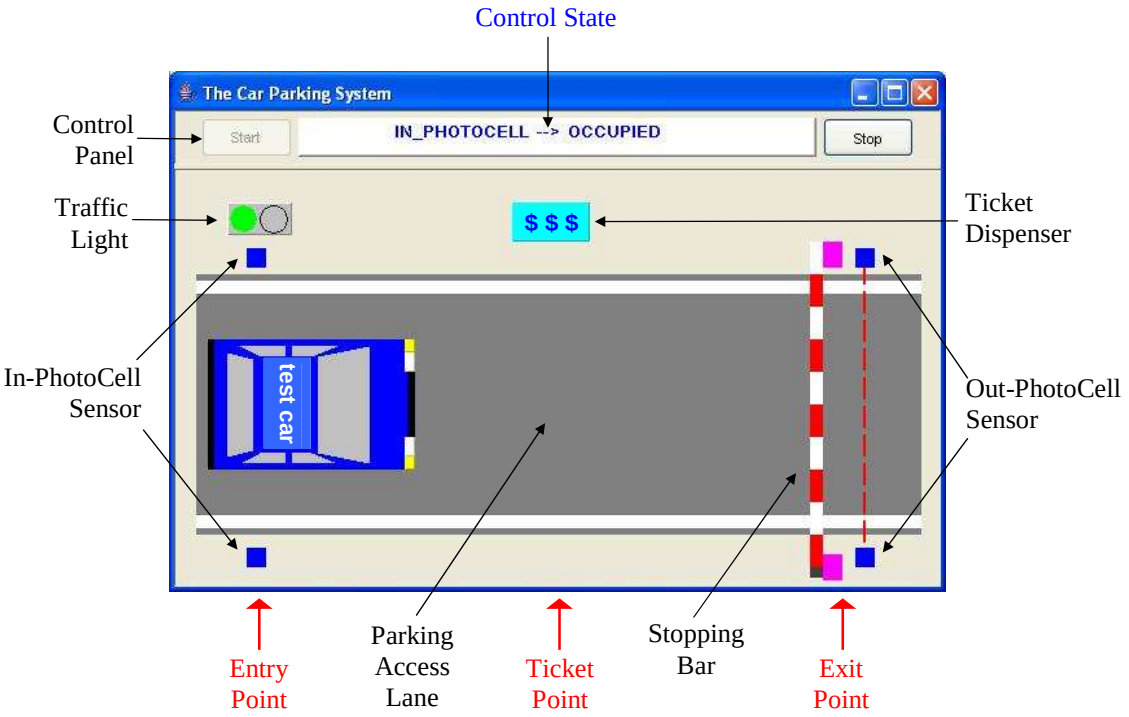


Figure B.1 The Car Parking System

(2) The second control point is the ticket point, which is controlled by the Ticket Dispenser device.

(a) The Ticket Dispenser device delivers a ticket to be withdrawn by the car driver.

- First, the Ticket Dispenser device delivers a parking ticket;
- Then, the current car driver withdraws the delivered ticket, which is used to pay a parking fare.

(3) The third control point is the exit point, which is jointly controlled by the Stopping Bar device and the Out-PhotoCell Sensor device.

(a) The Stopping Bar device controls a car's exiting the PAL exit point.

- The Stopping Bar raises up to allow the current car to exit the PAL exit point;
- The Stopping Bar lowers down after the current car has finished accessing the PAL exit point, or the Stopping Bar lowers down to disallow the current car to exit the PAL exit point.

(b) The Out-PhotoCell Sensor device senses whether or not the current car is accessing the PAL exit point.

- First, the Out-PhotoCell Sensor device senses that the PAL exit point has been occupied by the exiting car, when this car is accessing the PAL exit point;

- Then, the Out-PhotoCell Sensor device senses that the PAL exit point has been cleared by the same exiting car, after this car has finished accessing the PAL exit point.

B.2 Special Testing Requirements

In addition, the CPS system must be secure and reliable in order to provide high quality public access services. In the CPS case study, we have identified and examined a set of special quality requirements for supporting secure and reliable parking services. Among many other requirements, the following specifies a set of the three most important CPS special testing requirements (#1, #2, and #3), which become the principal testing and evaluation focus in the CPS case study.

(1) Special Testing Requirement #1: Parking Access Safety Rule

In the CPS system, all parking cars must abide by the parking access safety rule – "*one access at a time*", with the following specific mandatory public access requirements:

(a) Only one car can access the PAL (Parking Access Lane) at a time. This means that it is not allowed that two or more cars access the PAL at any same time.

(b) The next car is allowed to access the PAL only after the last car has finished its full PAL access.

This CPS safety rule is jointly supported by the correct control operations of the Traffic Light device and the In-PhotoCell Sensor device operated at the PAL entry point. This rule can prevent the occurrences of unsafe scenarios, e.g. possible car collisions due to multiple concurrent car accesses.

(2) Special Testing Requirement #2: Parking Pay-Service Rule

In the CPS system, all parking cars/drivers must comply with the parking pay-service rule – "*no pay, no parking*", with the specific requirement that the driver must withdraw a parking ticket to pay the required parking fare.

This CPS pay-service rule is mainly supported by the correct control operations of the Ticket Dispenser device operated at the PAL ticket point. This rule can assure the required level of financial support for public parking service operations.

(3) Special Testing Requirement #3: Parking Service Security Rule

In the CPS system, all parking cars must conform to this parking service security rule for any parking service violations, including:

(a) Violating the CPS safety rule

(b) Violating the CPS pay-service rule

(c) Any possible unsafe/insecure parking activities, e.g. excessive speeding along the PAL,

parking in unready/unavailable bays, parking in unauthorised areas, etc.

This CPS security rule is jointly supported by the correct control operations of the Stopping Bar device and the Out-PhotoCell Sensor device operated at the PAL exit point. This rule can assure the required level of public service safety/security protection and maintenance with the Stopping Bar device.

B.3 UML-Based Software Component Development

This section presents an overview of UML-based software component development for the CPS system. For this case study, we develop a software controller simulation for the CPS system, which simulates a typical public access control system, where a flow of cars and parking control devices are monitored, coordinated and regulated against certain public access requirements and rules (as shown earlier in Figure B.1). The CPS system is a typical reactive system: its dynamics are controlled and regulated by stimuli (events/actions) communicated with the external world (e.g. a parking user who is a car driver). Its main control structure for device communications employs an event-driven client-server control architecture. For event communications, we develop an independent, lightweight, base component EventCommunication, which is a pattern-based software component that is built on the Observer pattern [63] to implement a broadcaster-listener communication mechanism. Several application components are built on top of the Observer component that allows these components to work collaboratively to support event communications. The main application components include a device control component, a car control component and a GUI simulation component. The entire CPS system is componentised into a Java-based CBS.

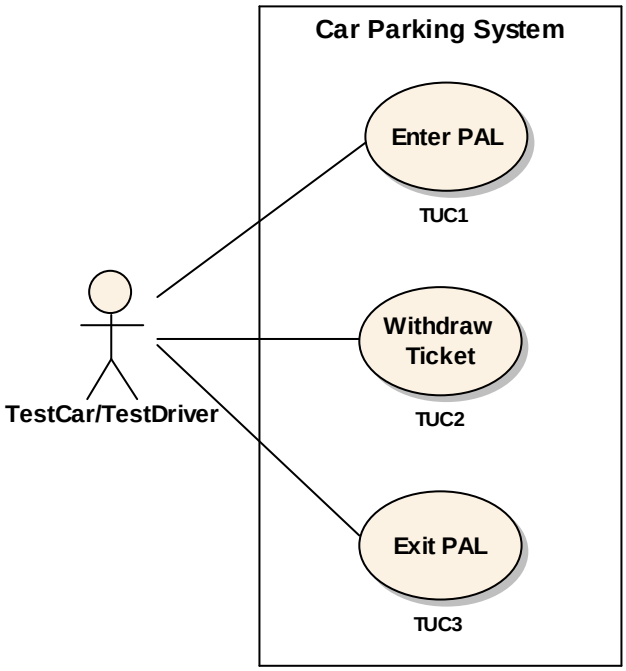
More details about the CPS system development are further described in [168], including UML-based component development and UML-based component specifications for the CPS system.

B.4 Constructing Test Models

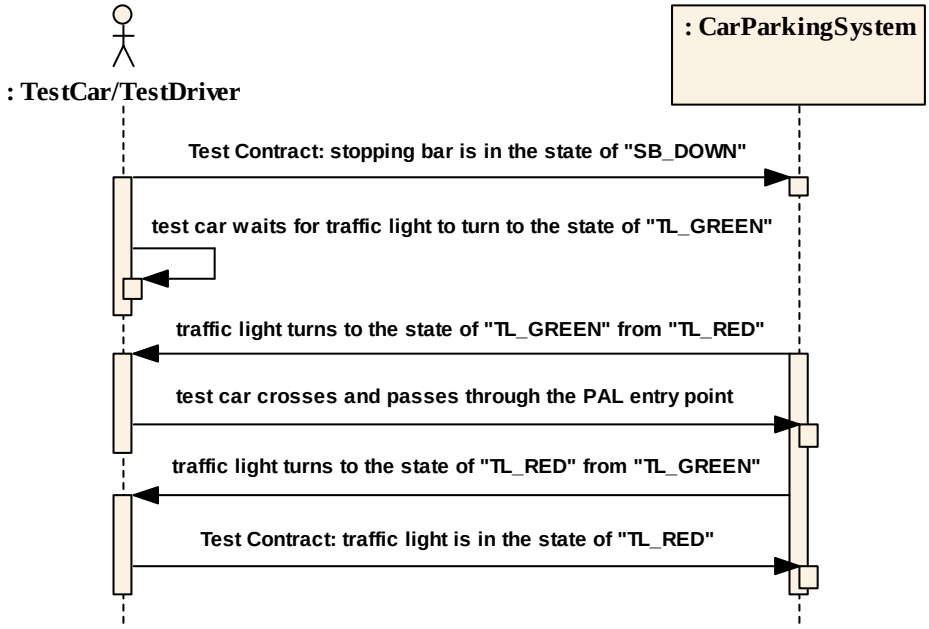
Chapter 4 to Chapter 5 have previously demonstrated the methodological characteristics and applicability of the MBSCT methodology and its framework for effective test model construction. Test model development is performed by applying the four main MBSCT methodological components: the model-based integrated SCT process, the scenario-based CIT technique, the TbC technique, and the TCR strategy. This section describes the construction of the use case test model (in Section B.4.1) and the design object test model (in Section B.4.2) undertaken in the CPS case study for the CIT purpose.

B.4.1 Use Case Test Model Construction

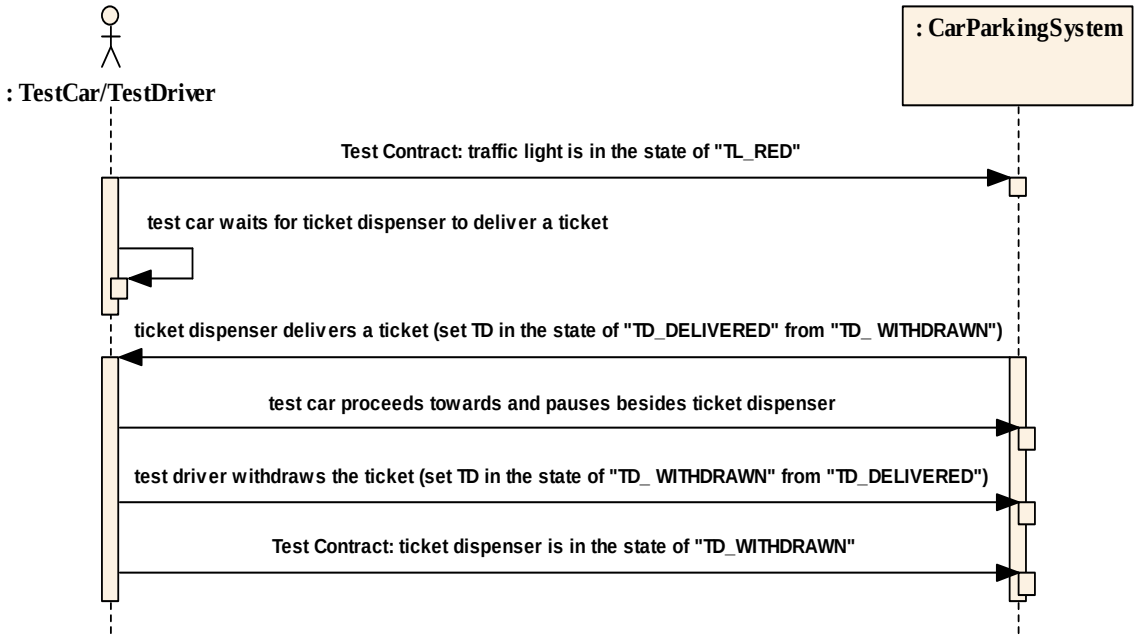
The use case test model (UCTM) for testing the CPS system was constructed as illustrated in Figure B.2. The UCTM is represented in four main parts: an overall test use case diagram shows the three core test use cases (TUCs) (as shown in Figure B.2 (a)), and three system test sequence diagrams show the main system-level test scenarios of the three individual CPS TUCs respectively (as illustrated in Figure B.2 (b), (c), (d)). In addition, as part of the UCTM, Table B.1 describes an overview of the three core CPS TUCs for testing the CPS system.



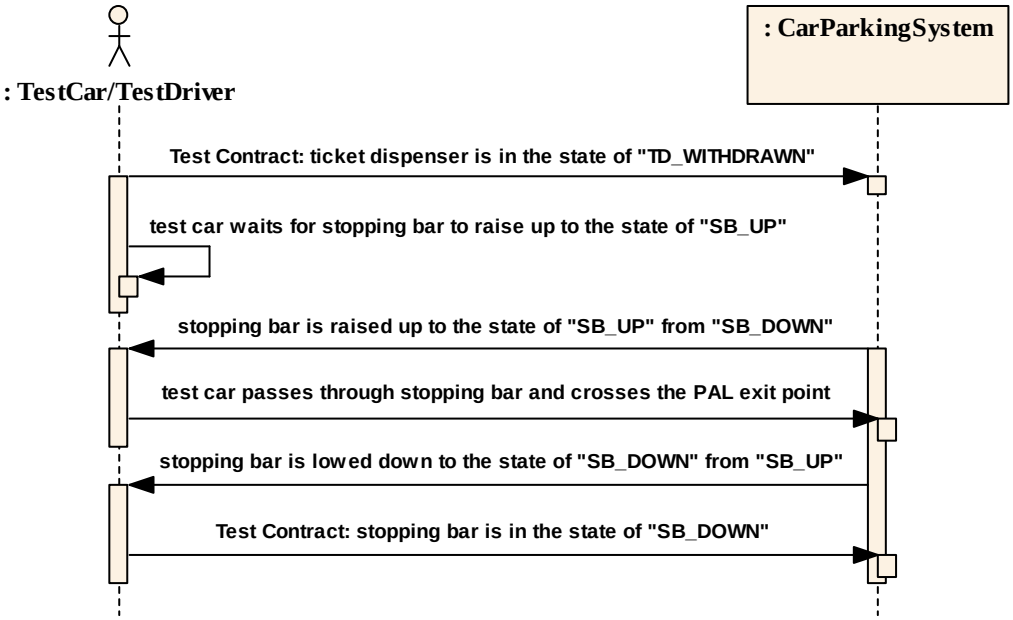
(a) Test Use Case Diagram (CPS System)



(b) System Test Sequence Diagram (CPS TUC1 Test Scenario)



(c) System Test Sequence Diagram (CPS TUC2 Test Scenario)



(d) System Test Sequence Diagram (CPS TUC3 Test Scenario)

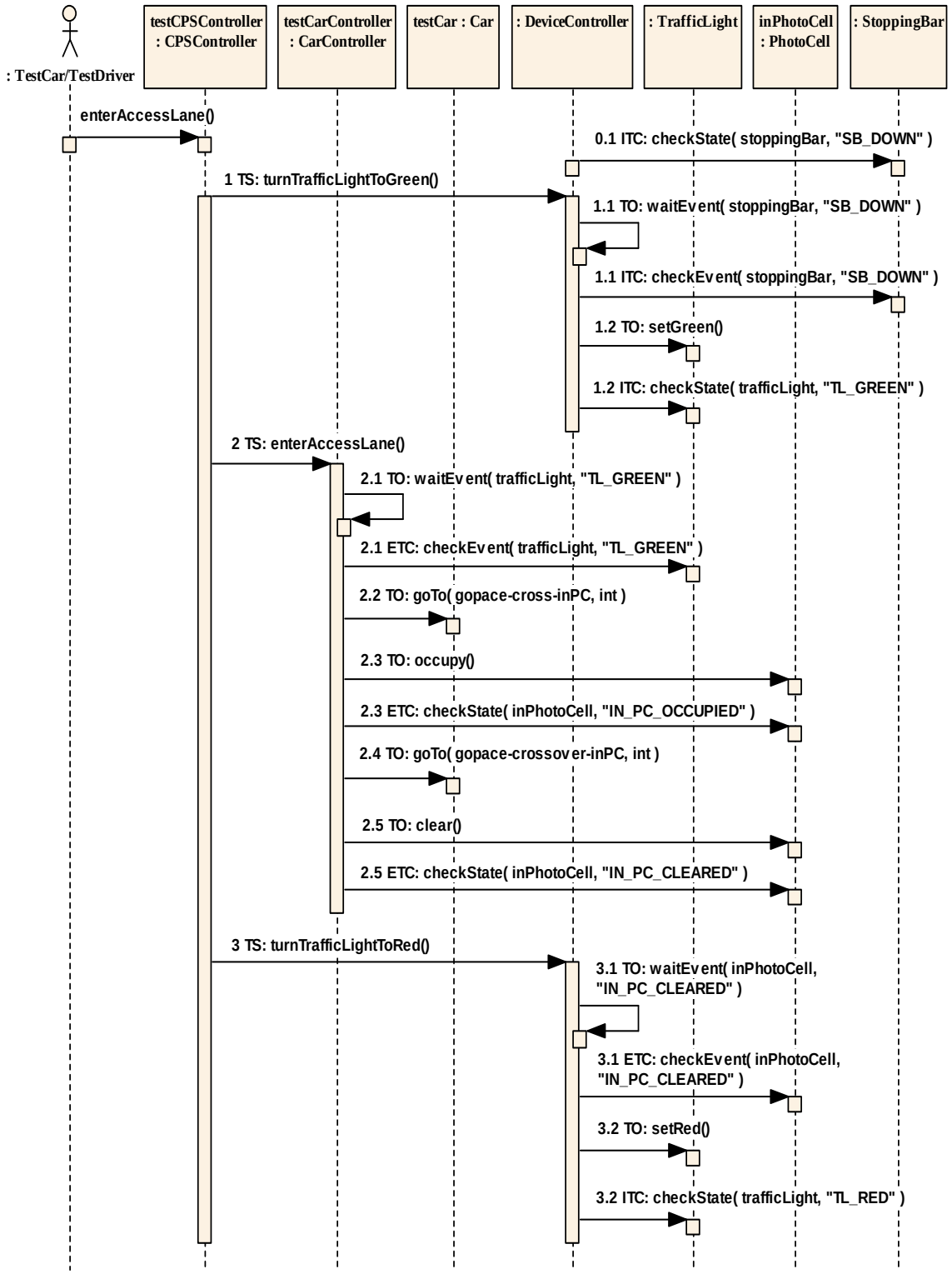
Figure B.2 Use Case Test Model (CPS System)

Table B.1 Use Case Test Model: Test Use Cases (CPS System)

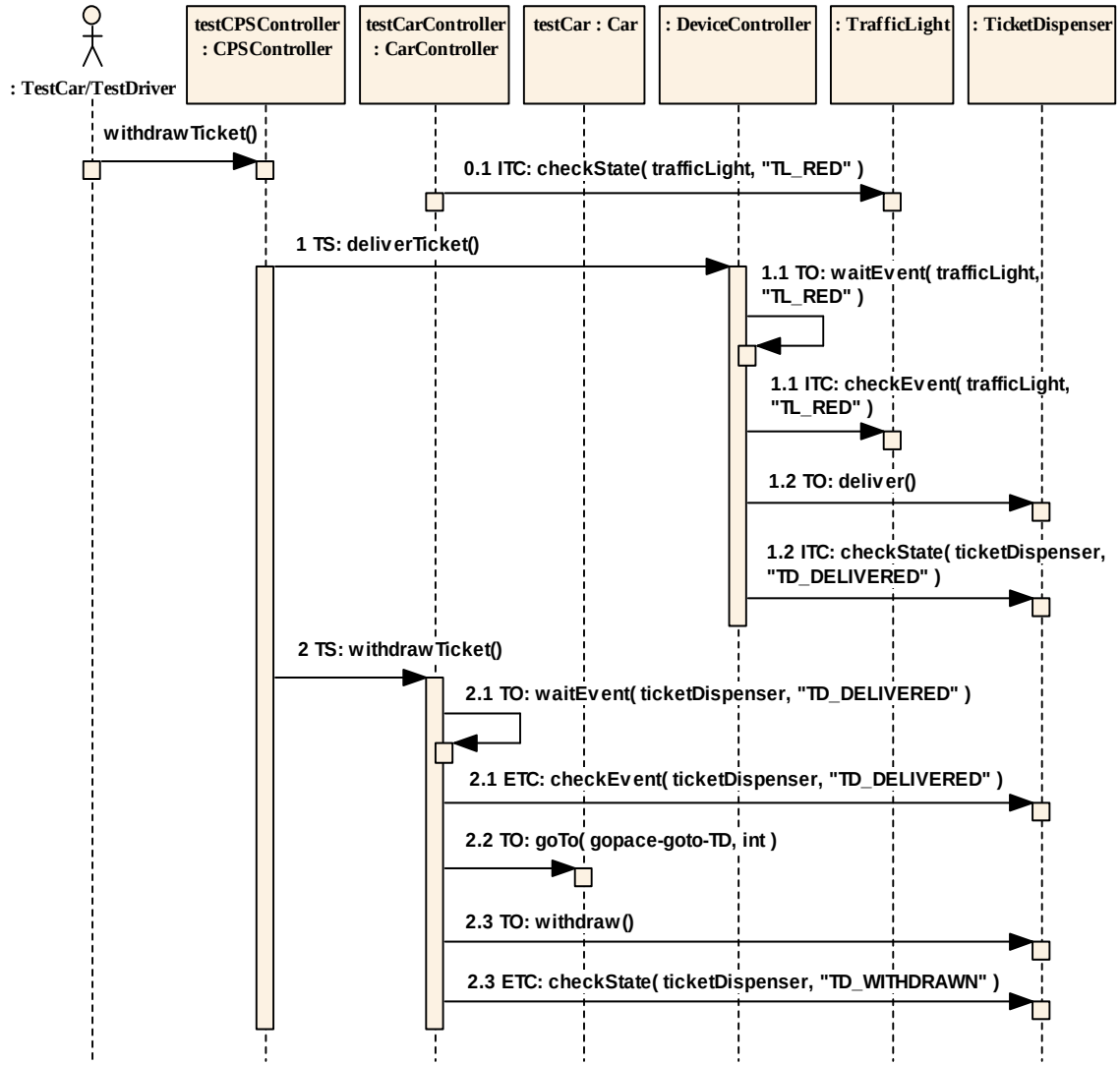
Test Use Case	Test Use Case Overview
CPS TUC1: Enter PAL	Exercise and examine that the test car enters the entry point of the parking access lane (PAL) to start accessing the PAL.
CPS TUC2: Withdraw Ticket	Exercise and examine that the test driver withdraws parking ticket at the PAL ticket point.
CPS TUC3: Exit PAL	Exercise and examine that the test car exits the PAL exit point to finish accessing the PAL.

B.4.2 Design Object Test Model Construction

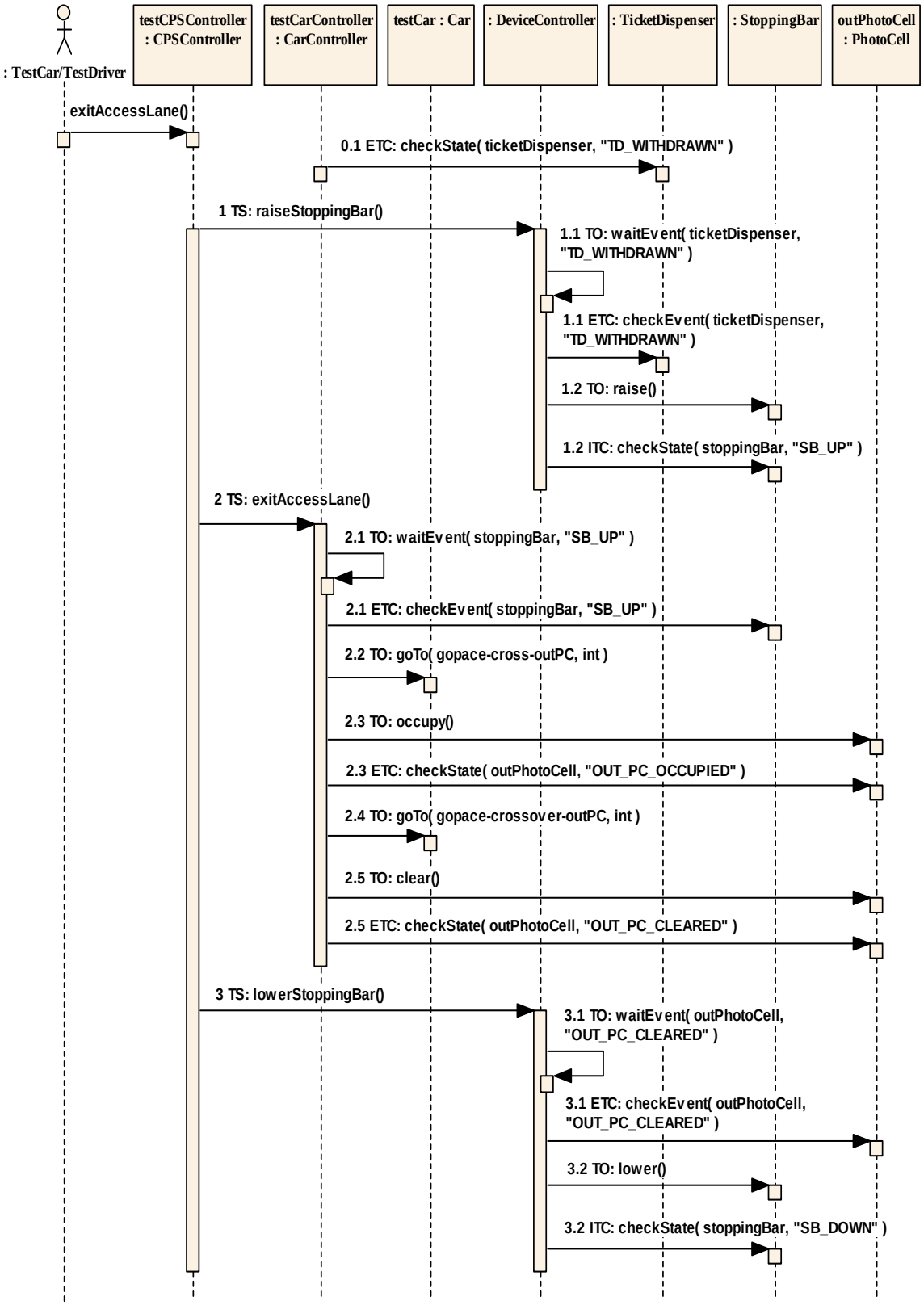
The design object test model (DOTM) for testing the CPS system was constructed as illustrated in Figure B.3. The DOTM is represented in three main parts: three test sequence diagrams show the main design-level test scenarios of the three individual CPS TUCs respectively (as illustrated in Figure B.3 (a), (b), (c)).



(a) Design Test Sequence Diagram (CPS TUC1 Test Scenario)



(b) Design Test Sequence Diagram (CPS TUC2 Test Scenario)



(c) Design Test Sequence Diagram (CPS TUC3 Test Scenario)

Figure B.3 Design Object Test Model (CPS System)

B.5 Designing and Generating Component Tests

Chapter 4 to Chapter 8 have previously illustrated and demonstrated the methodological characteristics and applicability of the MBSCT methodology and its framework for component test design and generation. Test case development is model-based, process-based, scenario-based, contract-based, FDD-based and mapping-based. This section describes component test derivation undertaken in the CPS case study for the CIT purpose.

Note that the description uses some naming conventions for acronyms or abbreviations of the following testing terms in the MBSCT methodology: TS – test sequence/scenario, TG – test group, TO – test operation, TC – test contract, and ITC/ETC – internal/external test contract.

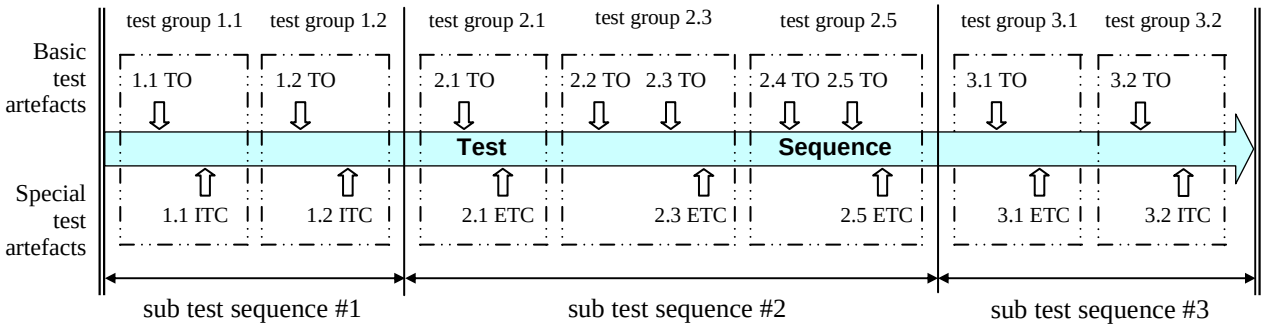
B.5.1 Test Sequence Design

The test sequence design for testing the CPS system was conducted to organise and structure a set of logically-ordered related test artefacts (e.g. test operations, test contracts and test elements) into test sequences for the three CPS TUC core test scenarios. The three main test sequences contain a total of eight (8) sub test sequences and a total of eighteen (18) test groups (as illustrated in Figure B.4).

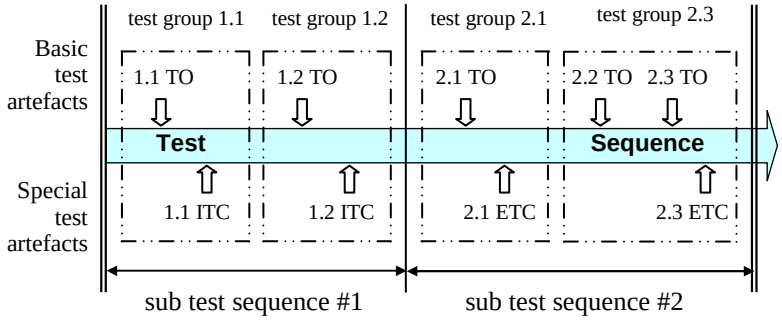
(a) The test sequence for the CPS TUC1 test scenario contains three (3) sub test sequences, where sub test sequence #1 comprises two (2) test groups, sub test sequence #2 comprises three (3) test groups, and sub test sequence #3 comprises two (2) test groups, with a subtotal of seven (7) test groups (as illustrated in Figure B.4 (a)).

(b) The test sequence for the CPS TUC2 test scenario contains two (2) sub test sequences, where sub test sequence #1 comprises two (2) test groups, and sub test sequence #2 comprises two (2) test groups, with a subtotal of four (4) test groups (as illustrated in Figure B.4 (b)).

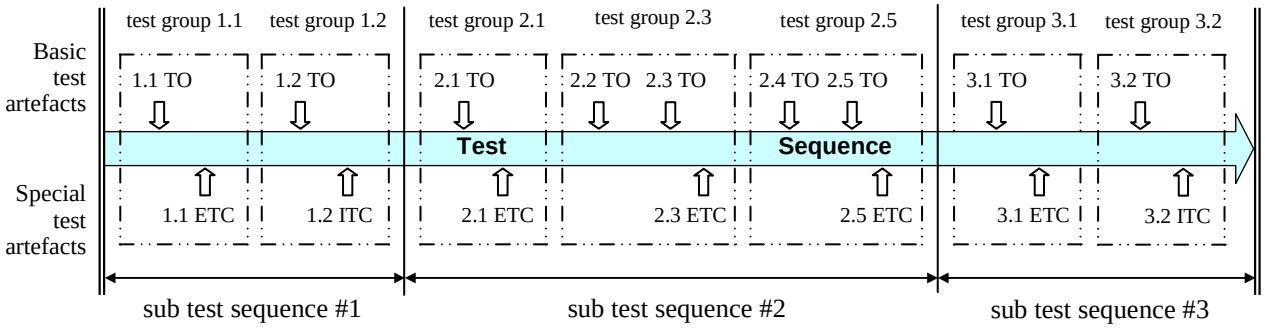
(c) The test sequence for the CPS TUC3 test scenario contains three (3) sub test sequences, where sub test sequence #1 comprises two (2) test groups, sub test sequence #2 comprises three (3) test groups, and sub test sequence #3 comprises two (2) test groups, with a subtotal of seven (7) test groups (as illustrated in Figure B.4 (c)).



(a) Structured Test Sequence (CPS TUC1 Test Scenario)



(b) Structured Test Sequence (CPS TUC2 Test Scenario)



(c) Structured Test Sequence (CPS TUC3 Test Scenario)

Figure B.4 Test Sequence Design (CPS System)

B.5.2 Component Test Design

Table B.2 shows the relationships between sub test sequences, test groups, test contracts, test operations (with specified signatures) and test states, which were used for component test design in the CPS case study for conducting CIT in the three CPS TUC core test scenarios (as illustrated in Table B.2 (a), (b) and (c) respectively).

Table B.2 Component Test Design (CPS System):
test sequences, test groups, test operations, test contracts and test states

Table B.2 (a) Component Test Design (CPS TUC1 Test Scenario)

Test Sequence	Test Group	Test Operation	Test Contract	Test State
enter PAL		enterAccessLane()		
			0.1 ITC: checkState(stoppingBar, "SB_DOWN")	SB_DOWN
Sub Test Sequence **#1** turn Traffic Light to GREEN		1 TS: turnTrafficLightToGreen()		
	1.1 TG	1.1 TO: waitEvent(stoppingBar, "SB_DOWN")	1.1 ITC: checkEvent(stoppingBar, "SB_DOWN")	SB_DOWN
	1.2 TG	1.2 TO: setGreen()	1.2 ITC: checkState(trafficLight, "TL_GREEN")	TL_GREEN
Sub Test Sequence **#2** enter the PAL entry point		2 TS: enterAccessLan()		
	2.1 TG	2.1 TO: waitEvent(trafficLight, "TL_GREEN")	2.1 ETC: checkEvent(trafficLight, "TL_GREEN")	TL_GREEN
	2.3 TG	2.2 TO: goTo(gopace-cross-inPC, int)		
		2.3 TO: occupy()	2.3 ETC: checkState(inPhotoCell, "IN_PC_OCCUPIED")	IN_PC_OCCUPIED
	2.5 TG	2.4 TO: goTo(gopace-crossover-inPC, int)		
		2.5 TO: clear()	2.5 ETC: checkState(inPhotoCell, "IN_PC_CLEARED")	IN_PC_CLEARED
Sub Test Sequence **#3** turn Traffic Light to RED		3 TS: turnTrafficLightToRed()		
	3.1 TG	3.1 TO: waitEvent(inPhotoCell, "IN_PC_CLEARED")	3.1 ETC: checkEvent(inPhotoCell, "IN_PC_CLEARED")	IN_PC_CLEARED
	3.2 TG	3.2 TO: setRed()	3.2 ITC: checkState(trafficLight, "TL_RED")	TL_RED

Table B.2 (b) Component Test Design (CPS TUC2 Test Scenario)

Test Sequence	Test Group	Test Operation	Test Contract	Test State
withdraw ticket		withdrawTicket()		
			0.1 ITC: checkState(trafficLight, "TL_RED")	TL_RED
Sub Test Sequence **#1** deliver ticket		1 TS: deliverTicket()		
	1.1 TG	1.1 TO: waitEvent(trafficLight, "TL_RED")	1.1 ITC: checkEvent(trafficLight, "TL_RED")	TL_RED
	1.2 TG	1.2 TO: deliver()	1.2 ITC: checkState(ticketDispenser, "TD_DELIVERED")	TD_DELIVERED
Sub Test Sequence **#2** withdraw ticket		2 TS: withdrawTicket()		
	2.1 TG	2.1 TO: waitEvent(ticketDispenser, "TD_DELIVERED")	2.1 ETC: checkEvent(ticketDispenser, "TD_DELIVERED")	TD_DELIVERED
	2.3 TG	2.2 TO: goTo(gopace-goto-TD, int)		
		2.3 TO: withdraw()	2.3 ETC: checkState(ticketDispenser, "TD_WITHDRAWN")	TD_WITHDRAWN

Table B.2 (c) Component Test Design (CPS TUC3 Test Scenario)

Test Sequence	Test Group	Test Operation	Test Contract	Test State
exit PAL		exitAccessLane()		
			0.1 ETC: checkState(ticketDispenser, "TD_WITHDRAWN")	TD_WITHDRAWN
Sub Test Sequence **#1** raise Stopping Bar up		1 TS: raiseStoppingBar()		
	1.1 TG	1.1 TO: waitEvent(ticketDispenser, "TD_WITHDRAWN")	1.1 ETC: checkEvent(ticketDispenser, "TD_WITHDRAWN")	TD_WITHDRAWN
	1.2 TG	1.2 TO: raise()	1.2 ITC: checkState(stoppingBar, "SB_UP")	SB_UP
Sub Test Sequence **#2** exit the PAL exit point		2 TS: exitAccessLane()		
	2.1 TG	2.1 TO: waitEvent(stoppingBar, "SB_UP")	2.1 ETC: checkEvent(stoppingBar, "SB_UP")	SB_UP
	2.3 TG	2.2 TO: goTo(gopace-cross-outPC, int)		
		2.3 TO: occupy()	2.3 ETC: checkState(outPhotoCell, "OUT_PC_OCCUPIED")	OUT_PC_OCCUPIED
	2.5 TG	2.4 TO: goTo(gopace-crossover-outPC, int)		
		2.5 TO: clear()	2.5 ETC: checkState(outPhotoCell, "OUT_PC_CLEARED")	OUT_PC_CLEARED
Sub Test Sequence **#3** lower Stopping Bar down		3 TS: lowerStoppingBar()		
	3.1 TG	3.1 TO: waitEvent(outPhotoCell, "OUT_PC_CLEARED")	3.1 ETC: checkEvent(outPhotoCell, "OUT_PC_CLEARED")	OUT_PC_CLEARED
	3.2 TG	3.2 TO: lower()	3.2 ITC: checkState(stoppingBar, "SB_DOWN")	SB_DOWN

B.5.3 Component Test Generation

This section shows the target CTS test case specifications that are derived in the CPS case study for the three CPS TUC core test scenarios, including:

(1) The CTS test case specification for the CPS TUC1 test scenario (as shown in Figure B.5)

(2) The CTS test case specification for the CPS TUC2 test scenario (as shown in Figure B.6)

(3) The CTS test case specification for the CPS TUC3 test scenario (as shown in Figure B.7)

```
... ... ... ...
<TestSpecification Name="CPS_TUC1_CTS.xml">
..<Desc>CTS test case specification for CPS TUC1: car enters PAL</Desc>
... ... ... ...

..<TestSet Name="TUC1_TestSet_turnTLtoGreen">
....<Desc>Test Set #1: this test set examines turning traffic light to the state
          of "TL_GREEN"</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>1.1 TG: grouped tests examine waiting the incoming event notified
            to turn traffic light</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>1.1 TO: examine waiting the incoming event notified to turn
              traffic light</Desc>
........<TestMethod Name="waitEvent" Target="deviceController">
..........<Desc>1.1 TO: deviceController waits the incoming event notification
                from stopping bar</Desc>
..........<Arg Name="aObservable" Source="stoppingBar"
                DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="SB_DOWN" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="deviceController">
..........<Desc>1.1 ITC: deviceController checks receiving the correct event
                notification from stopping bar</Desc>
..........<Arg Name="aObservable" Source="stoppingBar"
                DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="SB_DOWN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.1 ITC result: checkEvent must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="setGreen_groupedtests">
......<Desc>1.2 TG: grouped tests examine turning traffic light to the state
            of "TL_GREEN"</Desc>
......<TestOperation Name="setGreen_tests">
........<Desc>1.2 TO: examine turning traffic light to the state of "TL_GREEN"</Desc>
........<TestMethod Name="setGreen" Target="trafficLight">
..........<Desc>1.2 TO: turn traffic light to the state of "TL_GREEN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="trafficLight">
..........<Desc>1.2 ITC: check traffic light in the resulted correct state
                of "TL_GREEN"</Desc>
..........<Arg Name="aObservable" Source="trafficLight"
                DataType="java.util.Observable" />
..........<Arg Name="aState" Source="TL_GREEN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.2 ITC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC1_TestSet_carEnterPAL">
....<Desc>Test Set #2: this test set examines car entering PAL entry point</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>2.1 TG: grouped tests examine waiting the incoming event notified
            for car to enter PAL entry point</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>2.1 TO: examine waiting the incoming event notified for car to
              enter PAL entry point</Desc>
........<TestMethod Name="waitEvent" Target="testCarController">
..........<Desc>2.1 TO: testCarController waits the incoming event notification
                from traffic light</Desc>
..........<Arg Name="aObservable" Source="trafficLight"
```

```
              DataType="java.util.Observable" />
.........<Arg Name="aEvent" Source="TL_GREEN" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="testCarController">
.........<Desc>2.1 ETC: testCarController checks receiving the correct event
               notified from traffic light</Desc>
.........<Arg Name="aObservable" Source="trafficLight"
              DataType="java.util.Observable" />
.........<Arg Name="aEvent" Source="TL_GREEN" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>2.1 ETC result: checkEvent must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="occupy_groupedtests">
......<Desc>2.3 TG: grouped tests examine setting in-PhotoCell sensor in
           the state of "IN_PC_OCCUPIED"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.2 TO: examine the test car crossing PAL entry point</Desc>
........<TestMethod Name="goTo" Target="testCar">
.........<Desc>2.2 TO: the test car crosses PAL entry point controlled by
               in-PhotoCell sensor</Desc>
.........<Arg Name="gopace" Source="gopace-cross-inPC" DataType="int" />
........</TestMethod>
......</TestOperation>
......<TestOperation Name="occupy_tests">
........<Desc>2.3 TO: examine setting in-PhotoCell sensor in the state of
             "IN_PC_OCCUPIED"</Desc>
........<TestMethod Name="occupy" Target="inPhotoCell"
.........<Desc>2.3 TO: set in-PhotoCell sensor in the state of
               "IN_PC_OCCUPIED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="inPhotoCell">
.........<Desc>2.3 ETC: check in-PhotoCell sensor in the resulted correct
               state of "IN_PC_OCCUPIED"</Desc>
.........<Arg Name="aObservable" Source="inPhotoCell"
              DataType="java.util.Observable" />
.........<Arg Name="aState" Source="IN_PC_OCCUPIED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>2.3 ETC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="clear_groupedtests">
......<Desc>2.5 TG: grouped tests examine setting in-PhotoCell sensor in
           the state of "IN_PC_CLEARED"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.4 TO: examine the test car crosses over and passes through
             PAL entry point</Desc>
........<TestMethod Name="goTo" Target="testCar">
.........<Desc>2.4 TO: the test car crosses over and passes through PAL
               entry point</Desc>
.........<Arg Name="gopace" Source="gopace-crossover-inPC" DataType="int" />
........</TestMethod>
......</TestOperation>
......<TestOperation Name="clear_tests">
........<Desc>2.5 TO: examine setting in-PhotoCell sensor in the state
             of "IN_PC_CLEARED"</Desc>
........<TestMethod Name="clear" Target="inPhotoCell">
.........<Desc>2.5 TO: set in-PhotoCell sensor in the state of "IN_PC_CLEARED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="inPhotoCell">
.........<Desc>2.5 ETC: check in-PhotoCell sensor in the resulted correct
               state of "IN_PC_CLEARED"</Desc>
.........<Arg Name="aObservable" Source="inPhotoCell"
              DataType="java.util.Observable" />
.........<Arg Name="aState" Source="IN_PC_CLEARED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>2.5 ETC result: checkState must return true</Desc>
```

```
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC1_TestSet_turnTLtoRed">
....<Desc>Test Set #3: this test set examines turning traffic light to the state
          of "TL_RED"</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>3.1 TG: grouped tests examine waiting the incoming event notified
            to turn traffic light</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>3.1 TO: examine waiting the incoming event notified to turn
              traffic light</Desc>
........<TestMethod Name="waitEvent" Target="deviceController">
.........<Desc>3.1 TO: deviceController waits the incoming event notification
                from in-PhotoCell sensor</Desc>
.........<Arg Name="aObservable" Source="inPhotoCell"
               DataType="java.util.Observable" />
.........<Arg Name="aEvent" Source="IN_PC_CLEARED" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="deviceController">
.........<Desc>3.1 ETC: deviceController checks receiving the correct event
                notification from in-PhotoCell sensor</Desc>
.........<Arg Name="aObservable" Source="inPhotoCell"
               DataType="java.util.Observable" />
.........<Arg Name="aEvent" Source="IN_PC_CLEARED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>3.1 ETC result: checkEvent must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="setRed_groupedtests">
......<Desc>3.2 TG: grouped tests examine turning traffic light to the state
            of "TL_RED"</Desc>
......<TestOperation Name="setRed_tests">
........<Desc>3.2 TO: examine turning traffic light to the state of "TL_RED"</Desc>
........<TestMethod Name="setRed" Target="trafficLight">
.........<Desc>3.2 TO: turn traffic light to the state of "TL_RED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="trafficLight">
.........<Desc>3.2 ITC: check traffic light in the resulted correct state
                of "TL_RED"</Desc>
.........<Arg Name="aObservable" Source="trafficLight"
               DataType="java.util.Observable" />
.........<Arg Name="aState" Source="TL_RED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>3.2 ITC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure B.5 CTS Test Case Specification for the CPS TUC1 Test Scenario

```
... ... ... ...
<TestSpecification Name="CPS_TUC2_CTS.xml">
..<Desc>CTS test case specification for CPS TUC2: withdraw ticket</Desc>
... ... ... ...

..<TestSet Name="TUC2_TestSet_deliverTicket">
....<Desc>Test Set #1: this test set examines setting ticket dispenser in the state
          of "TD_DELIVERED"</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>1.1 TG: grouped tests examine waiting the incoming event notified
            to deliver ticket</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>1.1 TO: examine waiting the incoming event notified to
              deliver ticket</Desc>
........<TestMethod Name="waitEvent" Target="deviceController">
..........<Desc>1.1 TO: deviceController waits the incoming event notification
                from traffic light</Desc>
..........<Arg Name="aObservable" Source="trafficLight"
                DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="TL_RED" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="deviceController">
..........<Desc>1.1 ITC: deviceController checks receiving the correct event
                notification from traffic light</Desc>
..........<Arg Name="aObservable" Source="trafficLight"
                DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="TL_RED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.1 ITC result: checkEvent must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="deliver_groupedtests">
......<Desc>1.2 TG: grouped tests examine setting ticket dispenser in the state
            of "TD_DELIVERED"</Desc>
......<TestOperation Name=" deliver_tests">
........<Desc>1.2 TO: examine setting ticket dispenser in the state of
              "TD_DELIVERED"</Desc>
........<TestMethod Name="deliver" Target="ticketDispenser">
..........<Desc>1.2 TO: set ticket dispenser in the state of "TD_DELIVERED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="ticketDispenser">
..........<Desc>1.2 ITC: check ticket dispenser in the resulted correct state
                of "TD_DELIVERED"</Desc>
..........<Arg Name="aObservable" Source="ticketDispenser"
                DataType="java.util.Observable" />
..........<Arg Name="aState" Source="TD_DELIVERED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.2 ITC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC2_TestSet_withdrawTicket">
....<Desc>Test Set #2: this test set examines setting ticket dispenser in the state
          of "TD_WITHDRAWN"</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>2.1 TG: grouped tests examine waiting the incoming event notified
            to withdraw ticket</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>2.1 TO: examine waiting the incoming event notified to
              withdraw ticket</Desc>
........<TestMethod Name="waitEvent" Target="testCarController">
..........<Desc>2.1 TO: testCarController waits the incoming event notification
```

```
                from ticket dispenser</Desc>
..........<Arg Name="aObservable" Source="ticketDispenser"
               DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="TD_DELIVERED" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="testCarController">
..........<Desc>2.1 ETC: testCarController checks receiving the correct event
                notified from ticket dispenser</Desc>
..........<Arg Name="aObservable" Source="ticketDispenser"
               DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="TD_DELIVERED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.1 ETC result: checkEvent must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="withdraw_groupedtests">
......<Desc>2.3 TG: grouped tests examine setting ticket dispenser in the state
            of "TD_WITHDRAWN"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.2 TO: examine the test car crossing PAL ticket point</Desc>
........<TestMethod Name="goTo" Target="testCar">
..........<Desc>2.2 TO: the test car crosses PAL ticket point controlled by
                ticket dispenser</Desc>
..........<Arg Name="gopace" Source="gopace-goto-TD" DataType="int" />
........</TestMethod>
......</TestOperation>
......<TestOperation Name="withdraw_tests">
........<Desc>2.3 TO: examine setting ticket dispenser in the state of
              "TD_WITHDRAWN"</Desc>
........<TestMethod Name="withdraw" Target="ticketDispenser"
..........<Desc>2.3 TO: set ticket dispenser in the state of "TD_WITHDRAWN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="ticketDispenser">
..........<Desc>2.3 ETC: check ticket dispenser in the resulted correct
                state of "TD_WITHDRAWN"</Desc>
..........<Arg Name="aObservable" Source="ticketDispenser"
               DataType="java.util.Observable" />
..........<Arg Name="aState" Source="TD_WITHDRAWN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure B.6 CTS Test Case Specification for the CPS TUC2 Test Scenario

```
... ... ... ...
<TestSpecification Name="CPS_TUC3_CTS.xml">
..<Desc>CTS test case specification for CPS TUC3: car exits PAL</Desc>
... ... ... ...

..<TestSet Name="TUC3_TestSet_raiseStoppingBar">
....<Desc>Test Set #1: this test set examines raising up stopping bar to
          the state of "SB_UP"</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>1.1 TG: grouped tests examine waiting the incoming event notified
            to raise up stopping bar</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>1.1 TO: examine waiting the incoming event notified to
              raise up stopping bar</Desc>
........<TestMethod Name="waitEvent" Target="deviceController">
..........<Desc>1.1 TO: deviceController waits the incoming event notification
                from ticket dispenser</Desc>
..........<Arg Name="aObservable" Source="ticketDispenser"
               DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="TD_WITHDRAWN" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="deviceController">
..........<Desc>1.1 ITC: deviceController checks receiving the correct event
                notification from ticket dispenser</Desc>
..........<Arg Name="aObservable" Source="trafficLight"
               DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="TD_WITHDRAWN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.1 ITC result: checkEvent must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="raise_groupedtests">
......<Desc>1.2 TG: grouped tests examine raising up stopping bar to
            the state of "SB_UP"</Desc>
......<TestOperation Name="raise_tests">
........<Desc>1.2 TO: examine raising up stopping bar to the state of "SB_UP"</Desc>
........<TestMethod Name="raise" Target="stoppingBar">
..........<Desc>1.2 TO: raise up stopping bar to the state of "SB_UP"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="stoppingBar">
..........<Desc>1.2 ITC: check stopping bar in the resulted correct state
                of "SB_UP"</Desc>
..........<Arg Name="aObservable" Source="stoppingBar"
               DataType="java.util.Observable" />
..........<Arg Name="aState" Source="SB_UP" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.2 ITC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC3_TestSet_carExitPAL">
....<Desc>Test Set #2: this test set examines car exiting PAL exit point</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>2.1 TG: grouped tests examine waiting the incoming event notified
            for car to exit PAL exit point</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>2.1 TO: examine waiting the incoming event notified for car to
              exit PAL exit point</Desc>
........<TestMethod Name="waitEvent" Target="testCarController">
..........<Desc>2.1 TO: testCarController waits the incoming event notification
                from stopping bar</Desc>
..........<Arg Name="aObservable" Source="stoppingBar"
```

```
              DataType="java.util.Observable" />
.........<Arg Name="aEvent" Source="SB_UP" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="testCarController">
.........<Desc>2.1 ETC: testCarController checks receiving the correct event
               notified from stopping bar</Desc>
.........<Arg Name="aObservable" Source="stoppingBar"
              DataType="java.util.Observable" />
.........<Arg Name="aEvent" Source="SB_UP" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.1 ETC result: checkEvent must return true</Desc>
............<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="occupy_groupedtests">
......<Desc>2.3 TG: grouped tests examine setting out-PhotoCell sensor in
            the state of "OUT_PC_OCCUPIED"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.2 TO: examine the test car crossing PAL exit point</Desc>
........<TestMethod Name="goTo" Target="testCar">
.........<Desc>2.2 TO: the test car crosses PAL exit point controlled by
               out-PhotoCell sensor</Desc>
.........<Arg Name="gopace" Source="gopace-cross-outPC" DataType="int" />
........</TestMethod>
......</TestOperation>
......<TestOperation Name="occupy_tests">
........<Desc>2.3 TO: examine setting out-PhotoCell sensor in the state of
              "OUT_PC_OCCUPIED"</Desc>
........<TestMethod Name="occupy" Target="outPhotoCell"
.........<Desc>2.3 TO: set out-PhotoCell sensor in the state of
               "OUT_PC_OCCUPIED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="outPhotoCell">
.........<Desc>2.3 ETC: check out-PhotoCell sensor in the resulted correct
               state of "OUT_PC_OCCUPIED"</Desc>
.........<Arg Name="aObservable" Source="outPhotoCell"
              DataType="java.util.Observable" />
.........<Arg Name="aState" Source="OUT_PC_OCCUPIED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="clear_groupedtests">
......<Desc>2.5 TG: grouped tests examine setting out-PhotoCell sensor in
            the state of "OUT_PC_CLEARED"</Desc>
......<TestOperation Name="goTo_tests">
........<Desc>2.4 TO: examine the test car crosses over and passes through
              PAL exit point</Desc>
........<TestMethod Name="goTo" Target="testCar">
.........<Desc>2.4 TO: the test car crosses over and passes through PAL
               exit point</Desc>
.........<Arg Name="gopace" Source="gopace-crossover-outPC" DataType="int" />
........</TestMethod>
......</TestOperation>
......<TestOperation Name="clear_tests">
........<Desc>2.5 TO: examine setting out-PhotoCell sensor in the state
              of "OUT_PC_CLEARED"</Desc>
........<TestMethod Name="clear" Target="outPhotoCell">
.........<Desc>2.5 TO: set out-PhotoCell sensor in the state of
               "OUT_PC_CLEARED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="outPhotoCell">
.........<Desc>2.5 ETC: check out-PhotoCell sensor in the resulted correct
               state of "OUT_PC_CLEARED"</Desc>
.........<Arg Name="aObservable" Source="outPhotoCell"
              DataType="java.util.Observable" />
.........<Arg Name="aState" Source="OUT_PC_CLEARED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
```

```
............<Desc>2.5 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC3_TestSet_lowerStoppingBar">
....<Desc>Test Set #3: this test set examines lowering down stopping bar to
          the state of "SB_DOWN"</Desc>

....<TestGroup Name="waitEvent_groupedtests">
......<Desc>3.1 TG: grouped tests examine waiting the incoming event notified
            to lower down stopping bar</Desc>
......<TestOperation Name="waitEvent_tests">
........<Desc>3.1 TO: examine waiting the incoming event notified to lower down
              stopping bar</Desc>
........<TestMethod Name="waitEvent" Target="deviceController">
..........<Desc>3.1 TO: deviceController waits the incoming event notification
                from out-PhotoCell sensor</Desc>
..........<Arg Name="aObservable" Source="outPhotoCell"
               DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="OUT_PC_CLEARED" DataType="java.lang.Object" />
........</TestMethod>
........<TestMethod Name="checkEvent" Target="deviceController">
..........<Desc>3.1 ETC: deviceController checks receiving the correct event
                notification from out-PhotoCell sensor</Desc>
..........<Arg Name="aObservable" Source="outPhotoCell"
               DataType="java.util.Observable" />
..........<Arg Name="aEvent" Source="OUT_PC_CLEARED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>3.1 ETC result: checkEvent must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="lower_groupedtests">
......<Desc>3.2 TG: grouped tests examine lowering down stopping bar to the state
            of "SB_DOWN"</Desc>
......<TestOperation Name="lower_tests">
........<Desc>3.2 TO: examine lowering down stopping bar to the state of
              "SB_DOWN"</Desc>
........<TestMethod Name="raise" Target="stoppingBar">
..........<Desc>3.2 TO: lower down stopping bar to the state of "SB_DOWN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="stoppingBar">
..........<Desc>3.2 ITC: check stopping bar in the resulted correct state
                of "SB_DOWN"</Desc>
..........<Arg Name="aObservable" Source="stoppingBar"
               DataType="java.util.Observable" />
..........<Arg Name="aState" Source="SB_DOWN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>3.2 ITC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure B.7 CTS Test Case Specification for the CPS TUC3 Test Scenario

B.6 Evaluation Examples for Evaluating Adequate Test Artefact Coverage and Component Testability Improvement

In Chapter 9, Section 9.3.2 and Section 9.3.3 examine and evaluate the effectiveness of the MBSCT testing capabilities #4 and #5 (for adequate test artefact coverage and component testability improvement), specifically by using the first evaluation example for the CPS special testing requirement #1 in the CPS case study. This section presents the other two evaluation examples #2 and #3 for the two CPS special testing requirements #2 and #3 (in Subsections B.6.1 and B.6.2 respectively).

B.6.1 Evaluation Example #2: Parking Pay-Service Rule

The second evaluation example is about the CPS special testing requirement #2 (Parking Pay-Service Rule), and is related to the testing of the ticket dispenser device in the CPS TUC2 test scenario. The testing is also CIT-related, because the control operations of the ticket dispenser device are exercised and examined in the CPS TUC2 integration testing context.

The CPS system has a special testing requirement of the "*no pay, no parking*" rule for the purpose of financially-funded public service management (as described in Section B.2). For testing this CPS pay-service rule, the CPS test sequence design and component test design undertaken in the CPS case study (as described in Section B.5 above and Section 9.3.2 in Chapter 9) have provided adequate test artefact coverage for exercising and examining the testing-required control operations of the ticket dispenser device. The main test operations comprise 1.2 TO `deliver`() and 2.3 TO `withdraw`() in the CPS TUC2 test scenario, and they thus bridge *Test-Gap #1* (as described in Section 5.2.4.2 in Chapter 5). Furthermore, the CPS component test design constructs and applies appropriate test contracts to each of these testing-required control operations for testing the ticket dispenser device, and the main test contracts include 1.2 ITC `checkState`(`ticketDispenser`, "TD_DELIVERED") and 2.3 ETC `checkState`(`ticketDispenser`, "TD_WITHDRAWN"). This enables testing to evaluate relevant test results and obtain component testability improvement, which bridges *Test-Gap #2* (as described in Section 5.2.4.2 in Chapter 5). Thus, the CPS component test design can improve component testability and meet the CPS special testing requirement #2.

B.6.2 Evaluation Example #3: Parking Service Security Rule

The third evaluation example is about the CPS special testing requirement #3 (Parking Service Security Rule), and is related to the testing of the stopping bar device in the CPS TUC3 test sce-

nario. Similarly, since the control operations of the stopping bar device are exercised and examined in the CPS TUC3 integration testing context, the testing is CIT-related.

The CPS system has a special testing requirement of the "*public security protection and maintenance*" rule for the purpose of ensuring public service security (as described in Section B.2). For testing this CPS security rule, as described in Section B.5 above and Section 9.3.2 in Chapter 9, the CPS test sequence design and component test design undertaken in the CPS case study have provided adequate test artefact coverage for exercising and examining the testing-required control operations of the stopping bar device. The main test operations include 1.2 TO `raise`() and 3.2 TO `lower`() in the CPS TUC3 test scenario. Thus, these testing-required artefacts are capable of bridging *Test-Gap #1* (as described in Section 5.2.4.2 in Chapter 5). Moreover, the CPS component test design constructs and applies adequate test contracts to each of these testing-required control operations for testing the stopping bar device. The main test contracts comprise 1.2 ITC `checkState`(`stoppingBar`, "SB_UP") and 3.2 ITC `checkState`(`stoppingBar`, "SB_DOWN"). These testing-support artefacts enable testing to evaluate relevant test results and improve component testability, and thus bridge *Test-Gap #2* (as described in Section 5.2.4.2 in Chapter 5). Therefore, the CPS component test design can improve component testability and fulfil the CPS special testing requirement #3.

B.7 Evaluation Examples for Fault Case Scenario Analysis and Fault Diagnostic Solution Design

In Chapter 9, Section 9.3.4 examines and evaluates the effectiveness of the MBSCT testing capabilities #3 and #6 for fault detection, diagnosis and localisation, by conducting fault case scenario analysis and fault diagnostic solution design particularly with the first evaluation example for the CPS special testing requirement #1 in the CPS case study. For this FDD evaluation, this section describes the other two evaluation examples #2 and #3 for the two CPS special testing requirements #2 and #3 (in Subsections B.7.1 and B.7.2 respectively).

B.7.1 Evaluation Example #2: Parking Pay-Service Rule

(1) Fault Case Scenario and Analysis

For the major fault/failure scenario of the CPS pay-service rule: the test car crosses over the ticket point to move forward towards the PAL exit point, even though the test driver has not withdrawn the ticket for paying parking fare. The resulting failure is a pay-service violation of the "*no pay, no parking*" rule against the CPS special testing requirement #2.

(2) Fault-Related Test Scenario

This fault is related to the CPS TUC2 test scenario, where the fault diagnosis is CIT-related.

(3) Fault-Related Control Point

This fault is related to the CPS control point – the ticket point in the PAL.

(4) Fault-Related Control Device

This fault is related to the CPS control device – the ticket dispenser device, which is operated at the PAL ticket point.

(5) Direct Diagnostic Solution

The fault diagnostic solution with the CPS test design needs to comprise the following test groups in the CPS TUC2 test scenario:

(a) Test group 1.2 TG contains test operation 1.2 TO `deliver`() and its associated (post-condition) test contract 1.2 ITC `checkState`(`ticketDispenser`, "TD_DELIVERED"), and test state "TD_DELIVERED".

(b) Test group 2.3 TG contains test operation 2.3 TO `withdraw`() and its associated (post-condition) test contract 2.3 ETC `checkState`(`ticketDispenser`, "TD_WITHDRAWN"), and test state "TD_WITHDRAWN".

(6) Stepwise Diagnostic Solution

The fault diagnostic solution with the CPS TUC2 test design needs to comprise the following equivalent test artefacts as a special test group:

(a) Precondition: test contract TC_TD_DELIVERED, which functions equivalently to test contract 1.2 ITC in test group 1.2 TG in the CPS TUC2 test scenario.

(b) Test operation TO_TD_WITHDRAWN, which functions equivalently to test operation 2.3 TO in test group 2.3 TG in the CPS TUC2 test scenario.

(c) Postcondition: test contract TC_TD_WITHDRAWN, which functions equivalently to test contract 2.3 ETC in test group 2.3 TG in the CPS TUC2 test scenario.

B.7.2 Evaluation Example #3: Parking Service Security Rule

(1) Fault Case Scenario and Analysis

For the major fault/failure scenario of the CPS security rule: the stopping bar remains unlowered, even after the current car has finished its full access to the PAL (which means that the current car has already finished accessing the PAL exit point), or even if no car is accessing the PAL. The resulting failure is a security violation of the "*public security protection and maintenance*" rule against the CPS special testing requirement #3.

(2) Fault-Related Test Scenario

This fault is related to the CPS TUC3 test scenario, where the fault diagnosis is CIT-related.

(3) Fault-Related Control Point

This fault is related to the CPS control point – the exit point in the PAL.

(4) Fault-Related Control Device

This fault is related to the CPS control device – the stopping bar device, which is operated at the PAL exit point.

(5) Direct Diagnostic Solution

The fault diagnostic solution with the CPS test design needs to include the following test groups in the CPS TUC3 test scenario:

(a) Test group 1.2 TG contains test operation 1.2 TO `raise`() and its associated (postcondition) test contract 1.2 ITC `checkState`(`stoppingBar`, "SB_UP"), and test state "SB_UP".

(b) Test group 3.2 TG contains test operation 3.2 TO `lower`() and its associated (postcondition) test contract 3.2 ITC `checkState`(`stoppingBar`, "SB_DOWN"), and test state "SB_DOWN".

(6) Stepwise Diagnostic Solution

The fault diagnostic solution with the CPS TUC3 test design needs to include the following equivalent test artefacts as a special test group:

(a) Precondition: test contract TC_SB_UP, which functions equivalently to test contract 1.2 ITC in test group 1.2 TG in the CPS TUC3 test scenario.

(b) Test operation TO_SB_DOWN, which functions equivalently to test operation 3.2 TO in test group 3.2 TG in the CPS TUC3 test scenario.

(c) Postcondition: test contract TC_SB_DOWN, which functions equivalently to test contract 3.2 ITC in test group 3.2 TG in the CPS TUC3 test scenario.

B.8 Evaluation Examples for Evaluating Adequate Component Fault Coverage and Diagnostic Solutions and Results

In Chapter 9, Section 9.3.5 examines and evaluates the effectiveness of the MBSCT testing capability #6 for evaluating adequate component fault coverage and diagnostic solutions, particu-

larly with the first evaluation example for the CPS special testing requirement #1 in the CPS case study. For the further FDD evaluation here, this section shows the other two evaluation examples #2 and #3 for the two CPS special testing requirements #2 and #3 (in Subsections B.8.1 and B.8.2 respectively).

B.8.1 Evaluation Example #2: Parking Pay-Service Rule

This subsection diagnoses the possible directly and indirectly related faults that cause the major failure scenario of the CPS pay-service rule against the CPS special testing requirement #2. In the CPS case study, we developed and applied two individual fault diagnostic solutions (as described in Section B.7.1 and Table 9.3 in Chapter 9). Each fault diagnostic solution contained the relevant test groups in the CPS TUC2 test scenario for the CPS test design (as illustrated in Figure B.8 below).

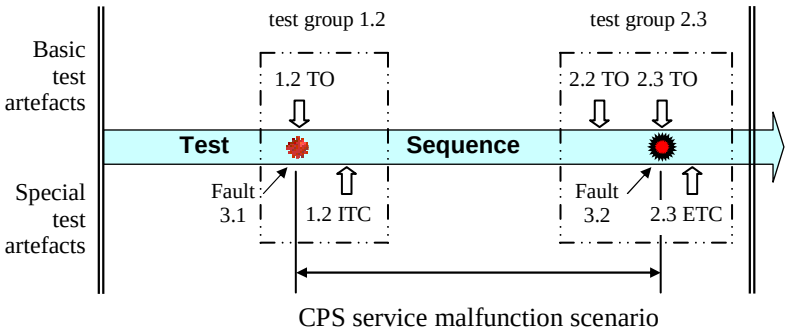


Figure B.8 Evaluation Example #2: Parking Pay-Service Rule (Fault Diagnostic Solutions with the CPS TUC2 Test Design)

The following describes our FDD evaluation for this major fault/failure scenario:

(1) Primary Fault 3.2 FAULT_TD_WITHDRAWN (as described in Table 9.3 in Chapter 9)

To diagnose the directly-related primary fault, the first fault diagnostic solution we developed is that the CPS TUC2 test design uses test group 2.3 TG to exercise test operation 2.3 TO `withdraw`(), which is verified by its associated (postcondition) test contract 2.3 ETC `checkState( ticketDispenser`, "TD_WITHDRAWN") and test state "TD_WITHDRAWN" in the CPS TUC2 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed the following fault: the fault is related to the ticket dispenser device operated at the PAL ticket point, where the ticket dispenser fails in the execution of operation `withdraw`(). This causes the ticket dispenser device NOT to be in the correct control state of "TD_WITHDRAWN" as expected, showing that the test driver has not withdrawn the ticket for paying the parking fare as expected. This is Primary Fault 3.2 FAULT_TD_WITHDRAWN as described in Table 9.3, which violates

the CPS pay-service rule ("*no pay, no parking*") against the CPS special testing requirement #2.

Thus, Primary Fault 3.2 FAULT_TD_WITHDRAWN directly results in the major fault/failure scenario of the CPS pay-service rule as described in Section B.7.1. The first fault diagnostic solution is able to diagnose this directly-related primary fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related operation `withdraw`() of the ticket dispenser device.

(2) Primary Fault 3.1 FAULT_TD_DELIVERED (as described in Table 9.3 in Chapter 9)

To diagnose an indirectly-related primary fault, the second fault diagnostic solution we developed employs test group 1.2 TG to exercise test operation 1.2 TO `deliver`(), which is verified by its associated (postcondition) test contract 1.2 ITC `checkState`(`ticketDispenser`, "TD_DELIVERED") and test state "TD_DELIVERED" in the CPS TUC2 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed a fault: the fault is related to the ticket dispenser device operated at the PAL ticket point, where the ticket dispenser fails in the execution of operation `deliver`(). This causes the ticket dispenser device NOT to be in the correct control state of "TD_DELIVERED" as expected, showing that the ticket dispenser fails to deliver a ticket to the test driver. This is Primary Fault 3.1 FAULT_TD_DELIVERED as described in Table 9.3. The occurrence of this fault could lead to a violated precondition, causing the test driver NOT to be able to withdraw the ticket for paying the parking fare as expected, i.e. the related succeeding operation `withdraw`() cannot be executed as expected or its execution fails.

Therefore, Primary Fault 3.1 FAULT_TD_DELIVERED could indirectly result in the occurrence of the major fault/failure scenario of the CPS pay-service rule as described in Section B.7.1. The second fault diagnostic solution is able to diagnose this indirectly-related primary fault. In the same way, following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is related to the ticket dispenser device's operation `deliver`() can be corrected and removed.

(3) Combined faults of the above two individual CPS primary faults

To diagnose the combined faults related to the ticket dispenser device's two operations, the fault diagnostic solution needs to combine the above two individual fault diagnostic solutions. Based on the above (1) to (2), the combined diagnostic solution can detect and diagnose the possible combinations of these two primary CPS faults, and the combined faults can be corrected and removed in the fault-related operations: the ticket dispenser device's operation `withdraw`() and/or operation `deliver`().

B.8.2 Evaluation Example #3: Parking Service Security Rule

This subsection diagnoses the possible directly and indirectly related faults causing the major failure scenario of the CPS security rule against the CPS special testing requirement #3. In the CPS case study, we developed and applied three individual fault diagnostic solutions (as described in Section B.7.2 and Table 9.3 in Chapter 9). Each fault diagnostic solution included the relevant test groups in the CPS TUC3 test scenario for the CPS test design (as illustrated in Figure B.9 below).

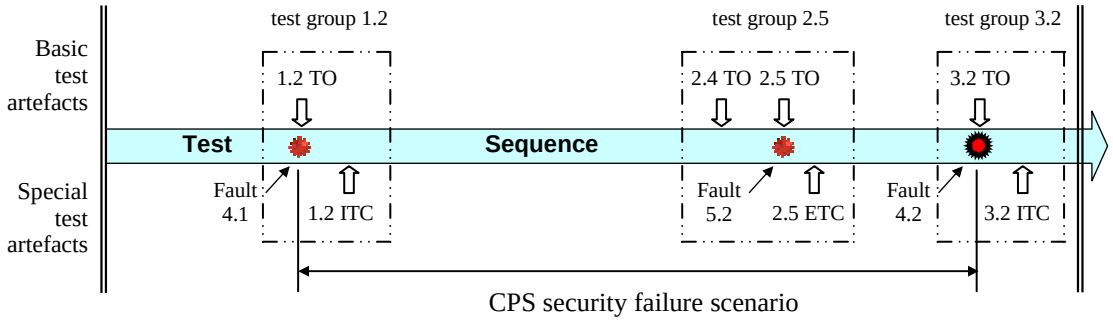


Figure B.9 Evaluation Example #3: Parking Service Security Rule (Fault Diagnostic Solutions with the CPS TUC3 Test Design)

Our FDD evaluation for this major fault/failure scenario is described as follows:

(1) Primary Fault 4.2 FAULT_SB_DOWN (as described in Table 9.3 in Chapter 9)

To diagnose the directly-related primary fault, the first fault diagnostic solution we developed is that the CPS TUC3 test design uses test group 3.2 TG to exercise test operation 3.2 TO `lower`(), which is verified by its associated (postcondition) test contract 3.2 ITC `checkState`(`stoppingBar`, "SB_DOWN") and test state "SB_DOWN" in the CPS TUC3 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed the following fault: the fault is related to the stopping bar device operated at the PAL exit point, where this CPS device fails in the execution of operation `lower`(). This causes the stopping bar device NOT to be in the correct control state of "SB_DOWN" as expected. This is Primary Fault 4.2 FAULT_SB_DOWN as described in Table 9.3, which results in a failure to abide by the CPS "*public security protection and maintenance*" rule against the CPS special testing requirement #3.

Hence, Primary Fault 4.2 FAULT_SB_DOWN directly causes the occurrence of the major fault/failure scenario of the CPS security rule as described in Section B.7.2. The first fault diagnostic solution is able to diagnose this directly-related primary fault. Following the CBFDD

guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related operation `lower`() of the stopping bar device.

(2) Primary Fault 4.1 FAULT_SB_UP (as described in Table 9.3 in Chapter 9)

To diagnose an indirectly-related primary fault, the second fault diagnostic solution we developed employs test group 1.2 TG to exercise test operation 1.2 TO `raise`(), which is verified by its associated (postcondition) test contract 1.2 ITC `checkState`(`stoppingBar`, "SB_UP")) and test state "SB_UP" in the CPS TUC3 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed a fault: the fault is related to the stopping bar device operated at the PAL exit point, where this CPS device fails in the execution of operation `raise`(), which causes the stopping bar device NOT to be in the correct control state of "SB_UP" as expected. This is Primary Fault 4.2 FAULT_SB_UP as described in Table 9.3. The occurrence of this fault indicates a violated precondition resulting from the preceding operation `raise`(); this violated precondition could cause the related succeeding operation `lower`() in the expected operation execution sequence NOT to be executed correctly, i.e. the stopping bar device's operation `lower`() cannot be executed as expected or its execution fails.

Thus, Primary Fault 4.2 FAULT_SB_UP could indirectly result in the occurrence of the major fault/failure scenario of the CPS security rule as described in Section B.7.2. The second fault diagnostic solution is able to diagnose this directly-related primary fault. In the same way, following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is related to operation `raise`() of the stopping bar device can be corrected and removed.

(3) Primary Fault 5.2 FAULT_OUT_PC_CLEARED (as described in Table 9.3 in Chapter 9)

For diagnosing an indirectly-related primary fault, the third fault diagnostic solution we developed uses test group 2.5 TG to exercise test operation 2.5 TO `clear`(), which is verified by its associated (postcondition) test contract 2.5 ETC `checkState`(`outPhotoCell`, "OUT_PC_CLEARED") and test state "OUT_PC_CLEARED" in the CPS TUC3 test scenario.

If the test contract returns *false*, the fault diagnostic solution has revealed a fault: the fault is related to the out-PhotoCell sensor device operated at the PAL exit point, where this CPS device fails in the execution of operation `clear`(), causing the out-PhotoCell sensor device NOT to be in the correct control state of "OUT_PC_CLEARED" as expected. This is Primary Fault 4.2 FAULT_OUT_PC_CLEARED as described in Table 9.3. The occurrence of this fault indicates that the current car might have not finished its access to the PAL exit point. Consequently, this fault could lead to a violated precondition resulting from the preceding operation `clear`();

this violated precondition could cause the related succeeding operation `lower`() in the expected operation execution sequence NOT to be executed correctly, i.e. the stopping bar device's operation `lower`() cannot be executed as expected or its execution fails.

Therefore, Primary Fault 4.2 FAULT_OUT_PC_CLEARED could indirectly result in the major fault/failure scenario of the CPS security rule as described in Section B.7.2. The third fault diagnostic solution is able to diagnose this indirectly-related primary fault. In the same manner, following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related operation `clear`() of the out-PhotoCell sensor device.

(4) Combined faults of the above three individual CPS primary faults

To diagnose the combined faults related to the stopping bar device and the out-PhotoCell sensor device, the fault diagnostic solution needs to combine the above three individual fault diagnostic solutions. Based on the above (1) to (3), the combined diagnostic solution can detect and diagnose the possible combinations of these three CPS primary faults, and the combined faults can be corrected and removed in the following fault-related operations:

(a) the stopping bar device's operation `lower`(), and/or
(b) the stopping bar device's operation `raise`(), and/or
(c) the out-PhotoCell sensor device's operation `clear`().

Appendix C
Case Study: Automated Teller Machine System

The testing of the Automated Teller Machine (ATM) system is the second major case study undertaken in this research, in order to further validate and evaluate the core MBSCT testing capabilities. Chapter 9 has presented the most important contents of the ATM case study. This appendix provides the background and complementary information about the ATM case study. The full ATM case study has been described earlier in [178].

C.1 Overview of the ATM System

This section presents an overview of the ATM system. The ATM example is a fairly very well-known case study in the area of object-oriented software development with the UML modeling and Unified Process. The ATM system used in our case study is based on a prototype example described in [124] [78], which is also used by many other researchers and authors in the literature. In our ATM case study, we present more comprehensive and rigorous descriptions of the UML-based software component development and testing for the ATM system [178]. Our case study particularly focuses on how the ATM system operates to provide the main banking services for the core ATM transactions, which are the most important functional operations and system requirements for the ATM system.

C.1.1 ATM Devices and Operations

The ATM system provides the typical ATM-based banking services for bank customers. The ATM system comprises a number of physical hardware devices that collaboratively work together to perform all ATM operations and controls, including ATM sessions, ATM transactions, ATM device operations and maintenances, etc.

The following describes the main ATM devices, relevant operations and functional requirements:

(1) Card Reader

The bank customer inserts an ATM card into the card slot of the Card Reader device, which reads in the card information (e.g. card number) encoded on the ATM card. Inserting a card activates a new transaction session. The bank system must validate the customer information (e.g. card number and PIN entered by the customer from the ATM Keypad device) before any subsequent ATM operation can be performed in any ATM transaction.

The Card Reader device can eject the inserted ATM card to the card slot when the bank customer finishes or cancels a transaction session. After the ejected card is taken away from the card slot by the customer, the current transaction session finishes. The Card Reader device can retain the inserted ATM card after the customer fails three times to enter a correct PIN (personal identification number).

(2) Customer Console: Keypad and Display

The Customer Console device is the interface between the bank customer and the ATM system, and contains the Keypad device and Display/Screen device. The ATM Keypad device allows the bank customer to enter the PIN (within the permitted three entry attempts) and the amount of money to be transacted, or enter other operation-required information, in order to perform appropriate transactions or operations. The ATM Display/Screen device shows a number of ATM operation menus/options, and allows the bank customer to select a type of transaction or bank account, or select other relevant ATM operations (e.g. cancel or select no more transactions).

(3) Cash Dispenser

The Cash Dispenser device, where cash notes are stored, dispenses multiple cash notes as requested by the bank customer to the cash dispensing slot for withdrawal by the customer during the “Withdraw Cash” transaction.

(4) Money Depositor

The bank customer deposits the money envelope (that contains cash notes or cheques to be deposited) into the Money Depositor device during the “Deposit Money” transaction. The money envelope is first dispensed to the envelope depositing slot; then the bank customer takes the money envelope, places the cash notes or cheques into the money envelope and inserts the money envelope into the envelope depositing slot for depositing money.

(5) Receipt Printer

The Receipt Printer device prints transaction receipts for the bank customer, who can get printed receipts from the receipt slot.

The ATM system communicates with the Bank ATM Server, whose main functions are to conduct relevant ATM-based banking operations, such as necessary bank account updating operations when an associated ATM transaction (e.g. the “Withdraw Cash” transaction in Section C.1.2) has finished, necessary bank validation operations to ensure that ATM transactions are performed correctly (in Section C.2), etc. As a part of the backend Bank system, the Bank ATM Server connects the ATM with the Bank system through network communication systems. The

overall ATM system comprises the ATM (in practice, a number of ATMs) and the Bank ATM Server (or the "Bank" for abbreviation). For simplicity in the current scope of this ATM case study, we do not cover all detailed operations about how the Bank system and the networked communication system work, as they simply provide the necessary supporting system services for the ATM system.

C.1.2 Core ATM Transactions

The ATM system provides a set of banking services to the bank's customers, and the following describes its four core ATM transactions:

(1) Inquire Balance

A bank customer can inquire about the available balance of any bank account linked to the ATM card. If the operation of customer validation fails, the customer cannot make an "Inquire Balance" transaction.

(2) Withdraw Cash

A bank customer can withdraw cash (e.g. multiple $20 cash notes) from any bank account linked to the ATM card. The withdraw-from account balance must be updated after withdrawing. If the customer validation operation fails or the operation of account balance validation fails, then the customer cannot make a "Withdraw Cash" transaction, the Cash Dispenser device does not dispense any cash and the withdraw-from account must remain unchanged.

(3) Deposit Money

A bank customer can deposit money (cash notes or cheques) into any bank account linked to the ATM card. The deposit-to account balance must be updated after depositing. If the operation of customer validation fails, then the customer cannot make a "Deposit Money" transaction, the un-deposited money must be returned to the customer and the deposit-to account must remain unchanged.

(4) Transfer Money

A bank customer can transfer money between any two bank accounts linked to the ATM card. Both the transfer-to account balance and the transfer-from account balance must be updated after transferring money. If the customer validation operation fails or the operation of transfer-from account balance validation fails, the customer cannot make a "Transfer Money" transaction and the two bank accounts remain unchanged.

The ATM system serves one bank customer at a time (i.e. one ATM session serves a single customer at a time), and the bank customer may select and perform one or more transactions

in an ATM session. A core ATM transaction describes a system integration scenario that controls a number of operations of the related ATM devices. Accordingly, these core ATM transactions are the primary basis for integration testing of the ATM system.

C.2 Special Testing Requirements

In addition to the above ATM system description in Section C.1, the ATM system must be secure and reliable for providing high quality banking services. In particular, we have identified and examined a set of special quality requirements for supporting secure and reliable banking services for the core ATM transactions in the ATM system. Accordingly, these special quality requirements become the most important ATM special testing requirements, which are regarded as the central focus of testing and evaluation undertaken in the ATM case study.

Among many other requirements, the following specifies a set of the eight most important special requirements (#1 to #8) of the ATM system. Note that the current scope of the ATM special testing requirements shown in this appendix mainly apply to the first two core ATM transactions “Inquire Balance” and “Withdraw Cash”. Other ATM special testing requirements applicable to the last two core ATM transactions are described in [178].

(1) Special Testing Requirement #1: Session Start Verification – verifying session started correctly

In the ATM system, a new ATM session starts with the customer inserting their ATM card into the Card Reader device. Session start verification has the following specific requirements:

(a) The new ATM session must be started correctly, which is confirmed by the examination of Special Testing Requirement #3: Customer Validation.

(2) Special Testing Requirement #2: Session Stop Verification – verifying session stopped/finished correctly

In the ATM system, an ATM session stops when the customer indicates that they have no more transactions in the current session, and finishes with the customer taking the ejected ATM card from the Card Reader device. Session stop verification has the following specific requirements:

(a) The current ATM session must be stopped correctly when the customer indicates that they have no more transactions to perform, which is confirmed by the examination that the ejected ATM card is taken away by the customer from the Card Reader device correctly.

(3) Special Testing Requirement #3: Customer Validation – validating the customer eligibil-

ity for accessing the ATM system

In the ATM system, a customer who wants to use the ATM must have an authorised ATM access permission. Because the ATM card represents the customer who accesses the ATM system, customer validation requires correct customer information (e.g. ATM card number and PIN) with the following specific requirements:

(a) The customer must have a valid ATM card, which must be correctly inserted into and read in by the Card Reader device. The ATM card information (e.g. card number encoded on the inserted card) being read in must be correct and identical to the card information stored in the bank system.

(b) The customer must have a valid PIN, which must be correctly entered into and read in from the Customer Console (Keypad) device. The PIN (personal identification number) being read in must be correct and identical to the PIN information stored in the bank system.

(c) Based on the above (a) and (b), the customer (or the ATM card representing the customer) must have authorised eligibility to access the ATM.

(4) Special Testing Requirement #4: Transaction Start Verification – verifying transaction started correctly

In the ATM system, a new ATM transaction starts with the customer selecting a type of transaction from the Customer Console (Display/Screen) device. Transaction start verification has the following specific requirements:

(a) The new ATM transaction must be started correctly, which is confirmed by the examination of Special Testing Requirement #6: Transaction Selection Validation.

(5) Special Testing Requirement #5: Transaction Stop Verification – verifying transaction stopped/finished correctly

In the ATM system, an ATM transaction stops when the current ATM transaction finishes with the customer taking the printed transaction receipt from the receipt slot of the Receipt Printer device. Transaction stop verification has the following specific requirements:

(a) The current ATM transaction must be stopped (or terminated) correctly, which is confirmed by the examination that the printed transaction receipt is taken away by the customer from the receipt slot of the Receipt Printer device correctly.

(6) Special Testing Requirement #6: Transaction Selection Validation – validating the customer-selected transaction access eligibility in the ATM system

In the ATM system, a customer (or the ATM card representing the customer) may be permitted to access certain types of ATM transactions available in the current ATM. For example, the “Deposit Money” transaction may be available on some selected ATMs in some se-

lected locations. Thus, transaction selection validation is necessary and has the following specific requirements:

(a) The type of the customer-selected transaction (e.g. “Withdraw Cash”) must be linked to the inserted ATM card in the current ATM under access.

(b) The selected ATM transaction type can be accessed by the customer for performing the selected ATM transaction.

(7) Special Testing Requirement #7: Account Selection Validation – validating the customer-selected account access permission in the ATM system

In the ATM system, an ATM card (which represents the customer who accesses the ATM system) is issued to be originally linked to the “Savings” account, and may not be permitted to access the “Cheque” or “Credit Card” account. Thus, account selection validation is necessary and has the following specific requirements:

(a) The type of the customer-selected account (e.g. “Savings” account) must be valid for the customer’s account in the bank system.

(b) The type of the customer-selected account (e.g. “Savings” account) must be linked to the inserted ATM card in the current ATM under access.

(c) This selected account in the bank system can be accessed by the customer for performing the customer-selected ATM transaction.

(8) Special Testing Requirement #8: Account Balance Validation – validating the available credit balance of the customer-selected account that can be transacted correctly in the ATM system

In the ATM system, the customer-selected account must have a sufficient credit balance available for correctly performing certain ATM transactions, such as “Withdraw Cash” or “Transfer Money”. Account balance validation has the following specific requirements:

(a) The customer-selected account must have previously been validated correctly as described in the above “Special Testing Requirement #7: Account Selection Validation”.

(b) The available credit balance of the customer-selected account must be sufficient, and must be greater than or equal to the transaction amount (i.e. the customer-requested amount of money that can be transacted correctly in the customer-selected ATM transaction).

C.3 UML-Based Software Component Development

This section describes an overview of UML-based software component development for the ATM system. For this case study, we develop a software controller simulation for the ATM sys-

tem, which simulates the core ATM transactions performed with a set of ATM devices and operations in the ATM system. The ATM system simulation is developed with object-oriented component development, UML modeling and the Unified Process, which produces a number of UML-based software models as the main component specifications for UML-based software component development. The ATM system is componentised into a Java-based CBS for the purpose of SCT with the MBSCT methodology and its framework. The five main application components comprise ATM Session, ATM Transaction, ATM Devices, ATM GUI and Bank. More details about the ATM system development are further described in [178].

C.4 Constructing Test Models

The testing of the ATM system starts with building UML-based test models. In the ATM case study, we apply the four main MBSCT methodological components for test model development: the model-based integrated SCT process, the scenario-based CIT technique, the TbC technique and the TCR strategy (as described earlier in Chapter 4 to Chapter 5). As the illustrative examples for the purpose of model-based CIT of the ATM system, this section describes the development of the use case test model (in Section C.4.1) and the design object test model (in Section C.4.2) for the ATM case study.

C.4.1 Use Case Test Model Construction

This section describes the use case test model (UCTM) constructed for the ATM case study. Figure C.1 illustrates the test use case diagram (including the main test use cases and sub test use cases), and Table C.1 describes an overview of these test use cases. The ATM UCTM employs a <<include>> relationship between the including test use case "Perform Session" and the included test use case "Perform Transaction". The ATM Session test use case has two *session-specific* sub test use cases ("Start Session" and "Stop Session"), where a specific ATM transaction is exercised and examined in between these two sub test use cases. In addition, the ATM UCTM shows a *generalisation* relationship between the general (or abstract) test use case "Perform Transaction" and the specialised (or concrete) test use case for each of the four core ATM transactions, which are identified as the *core test use cases* (TUCs). Each TUC is *transaction-specific* and can be examined independently for the CIT purpose.

The ATM case study presented in this thesis focuses on the testing of the ATM Session and the first two ATM TUCs (i.e. ATM TUC1 and ATM TUC2). As part of the ATM UCTM, the three system test sequence diagrams are created for the ATM Session test scenario (as illustrated in Figure C.2), the ATM TUC1 core test scenario (as illustrated in Figure C.3), and the ATM TUC2 core test scenario (as illustrated in Figure C.4). Each system test sequence diagram

shows a sequence of main system test messages/events and the overall test contracts of the related ATM test scenario. Note that the later core test use case scenarios cover the *transaction-specific* core test scenarios for the ATM TUC1 and TUC2, but do not include the two sub test scenarios ("Start Session" and "Stop Session") that are separately described in the *session-specific* test scenario of the ATM Session test use case.

Note that the Bank (as shown in Figure C.1) represents a part of the backend Bank system, the Bank ATM Server, which is mainly responsible for ATM-based banking operations (e.g. necessary bank validation operations as described in Section C.2). It connects the ATM with the Bank system through network communication systems to provide the necessary supporting system services for the ATM system. The overall ATM system comprises the ATM and the Bank ATM Server (or the "Bank" for abbreviation). We also use the "ATM/Bank" system when dealing with some operations that are related to a specific ATM device or a specific Bank operation.

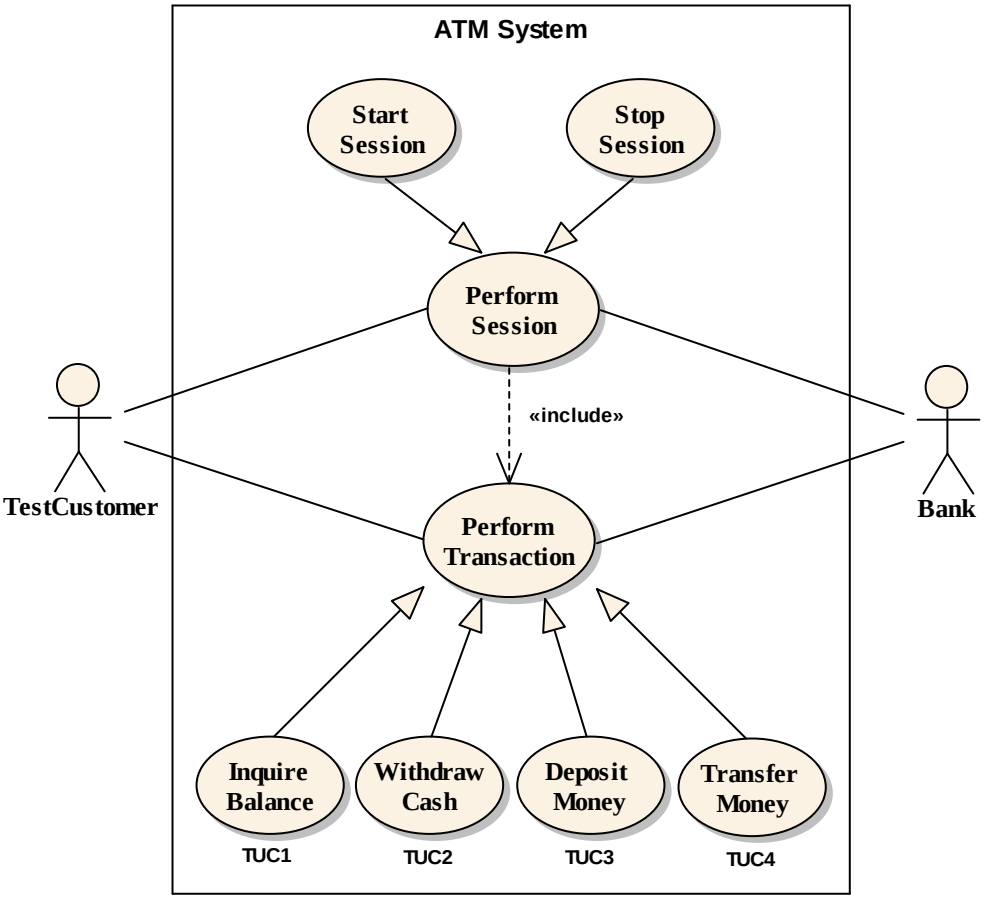


Figure C.1 Use Case Test Model: Test Use Case Diagram (ATM System)

Table C.1 Use Case Test Model: Test Use Cases (ATM System)

Test Use Case	Sub Test Use Case	Test Use Case Overview
Perform Session		Exercise and examine that a bank customer performs (start and stop) an ATM session for performing ATM transactions.
	Start Session	Exercise and examine that the bank customer starts an ATM session to perform one or more ATM transactions.
	Stop Session	Exercise and examine that the bank customer stops the current ATM session when indicating no more transaction.
Perform Transaction		Exercise and examine that the bank customer performs (start, do and stop) a specific ATM transaction within the ATM session.
	ATM TUC1: Inquire Balance	Exercise and examine that the bank customer inquires about the available balance of the any bank account (e.g. “Savings” account) linked to the ATM card.
	ATM TUC2: Withdraw Cash	Exercise and examine that the bank customer withdraws the requested amount of cash notes (e.g. multiple $20 notes) from any bank account (e.g. “Savings” account) linked to the ATM card.
	ATM TUC3: Deposit Money	Exercise and examine that the bank customer deposits money (cash notes or cheques) into any bank account (e.g. “Savings” account) linked to the ATM card.
	ATM TUC4: Transfer Money	Exercise and examine that the bank customer transfers money between any two bank accounts (e.g. from “Savings” account to “Cheque” account) linked to the ATM card.

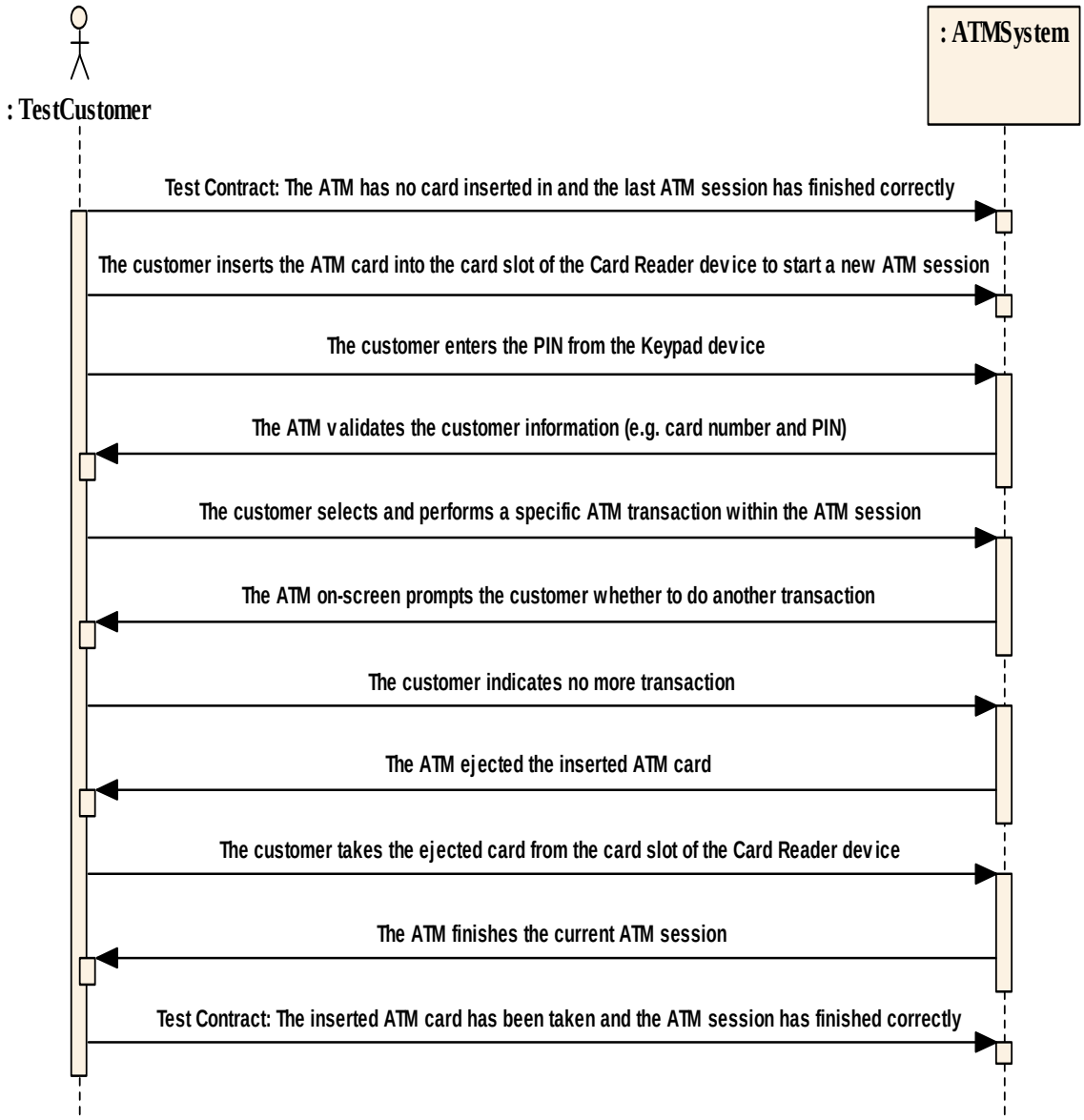


Figure C.2 Use Case Test Model: System Test Sequence Diagram
(ATM Session Test Scenario)

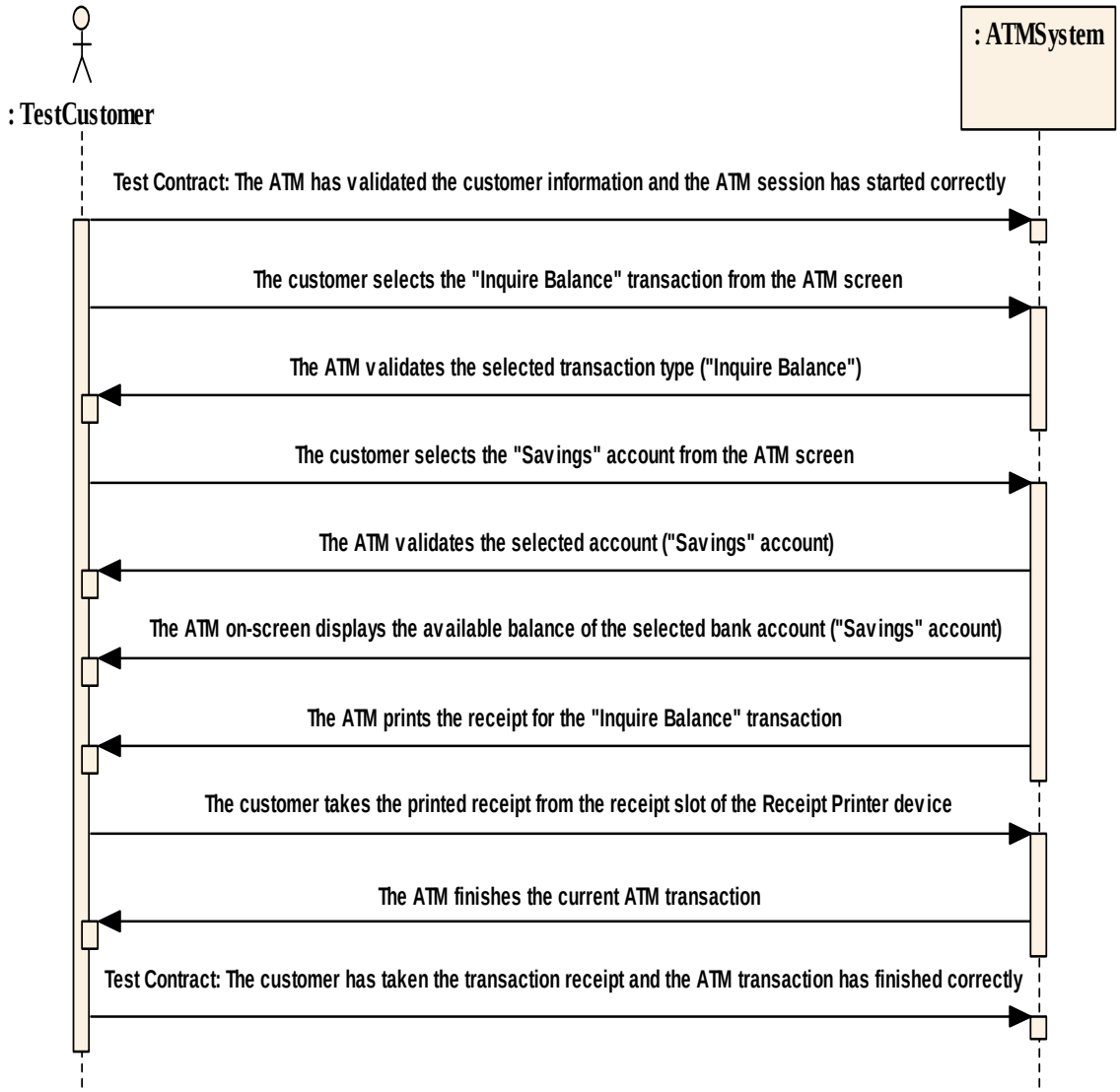


Figure C.3 Use Case Test Model: System Test Sequence Diagram
(ATM TUC1 Core Test Scenario)

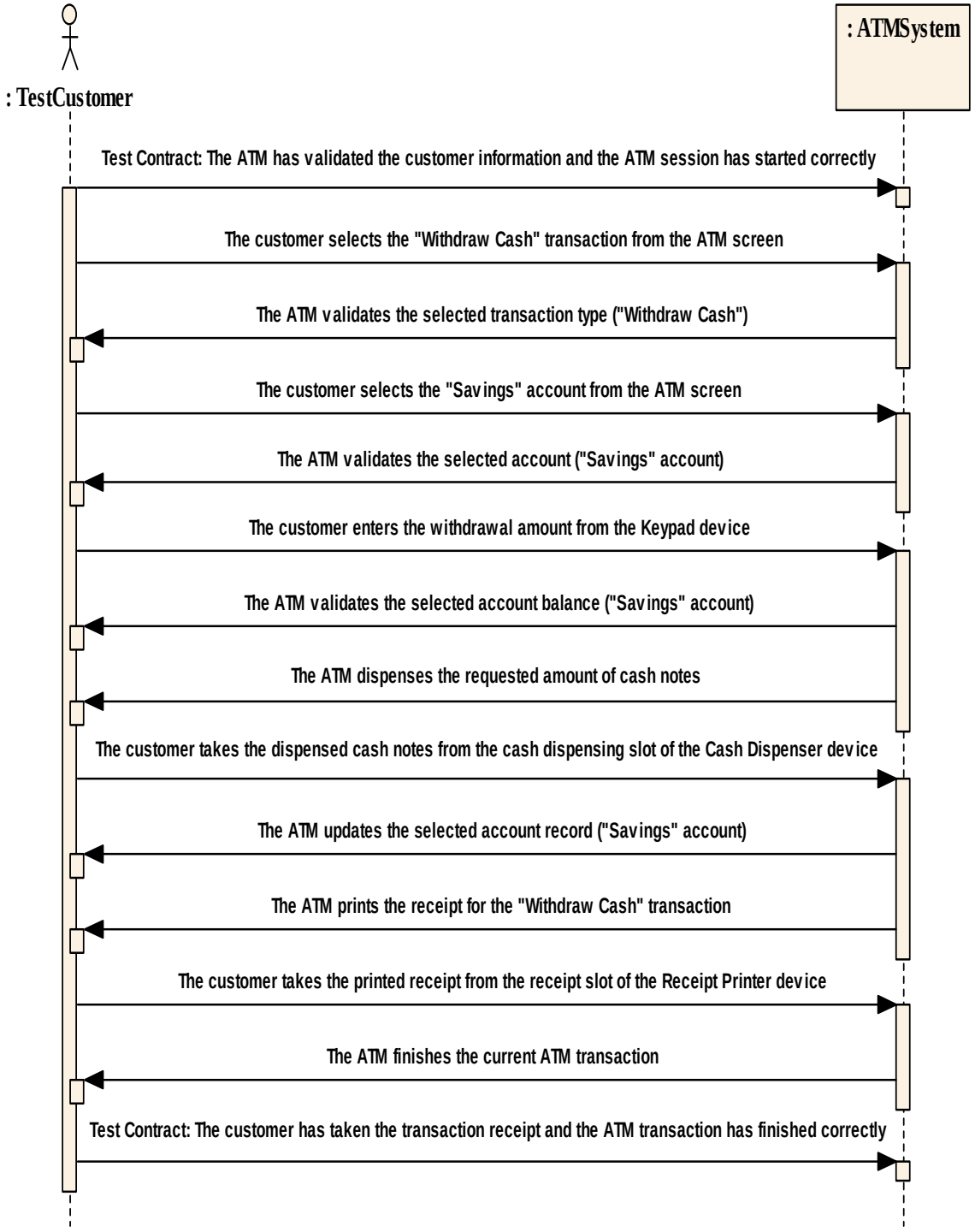


Figure C.4 Use Case Test Model: System Test Sequence Diagram
(ATM TUC2 Core Test Scenario)

C.4.2 Design Object Test Model Construction

This section presents the design object test model (DOTM) constructed in the ATM case study. The DOTM is mainly described with design test sequence diagrams to illustrate design test sequences, design test messages/operations and associated test contracts that jointly realise the ATM test use cases described in the ATM UCTM, as shown in Figure C.5 to Figure C.7.

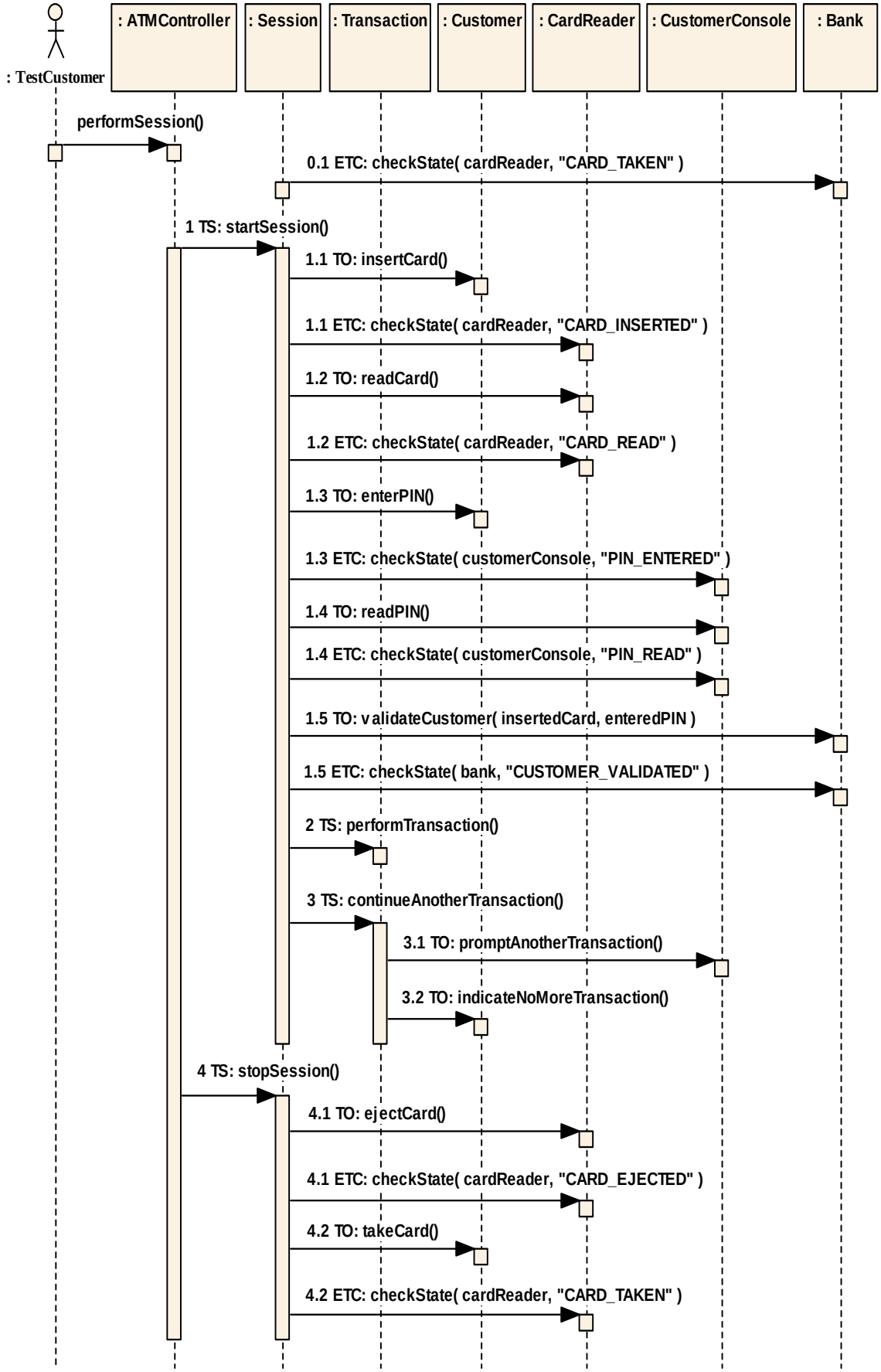


Figure C.5 Design Object Test Model: Design Test Sequence Diagram (ATM Session Test Scenario)

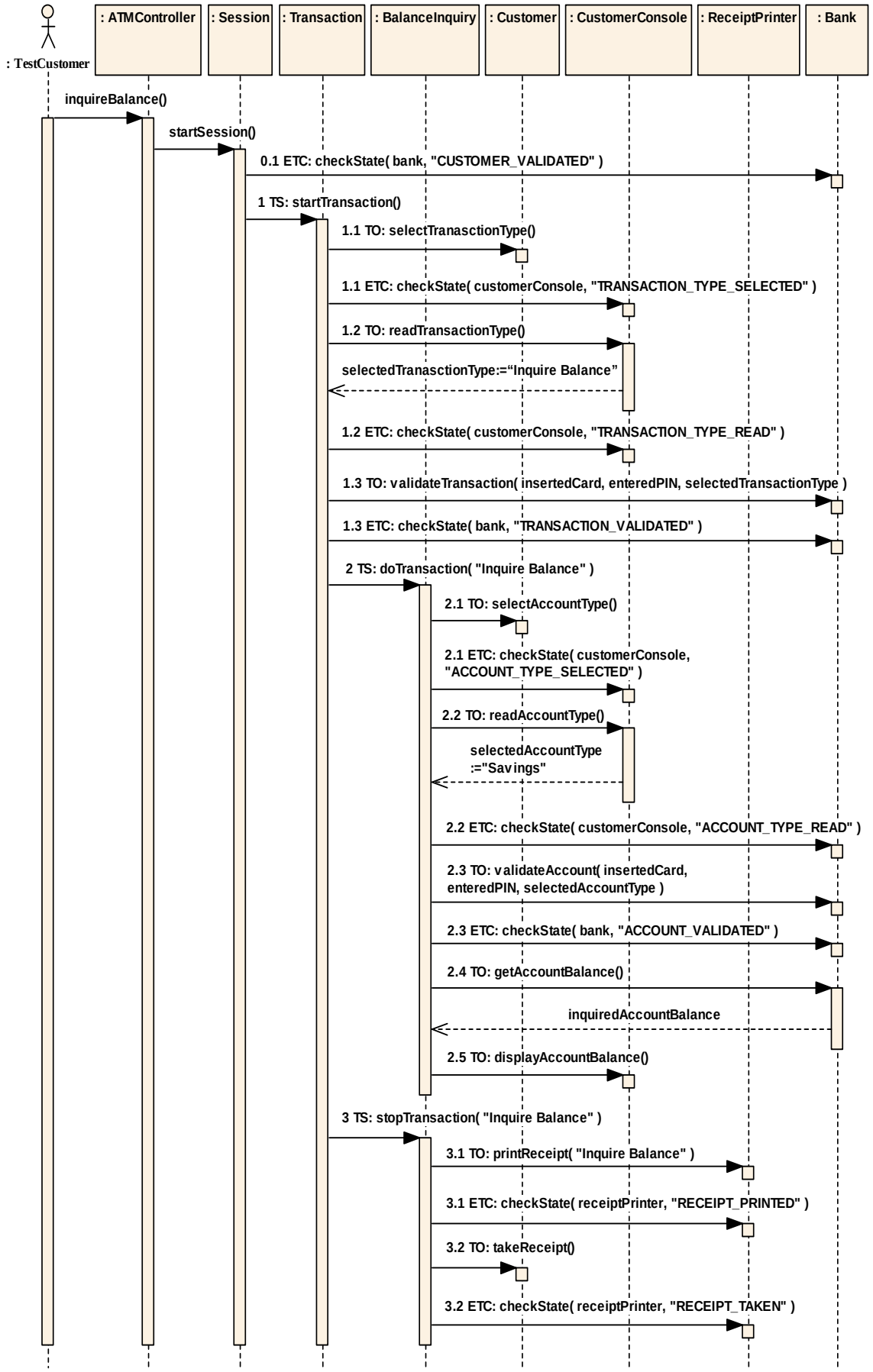


Figure C.6 Design Object Test Model: Design Test Sequence Diagram
(ATM TUC1 Core Test Scenario)

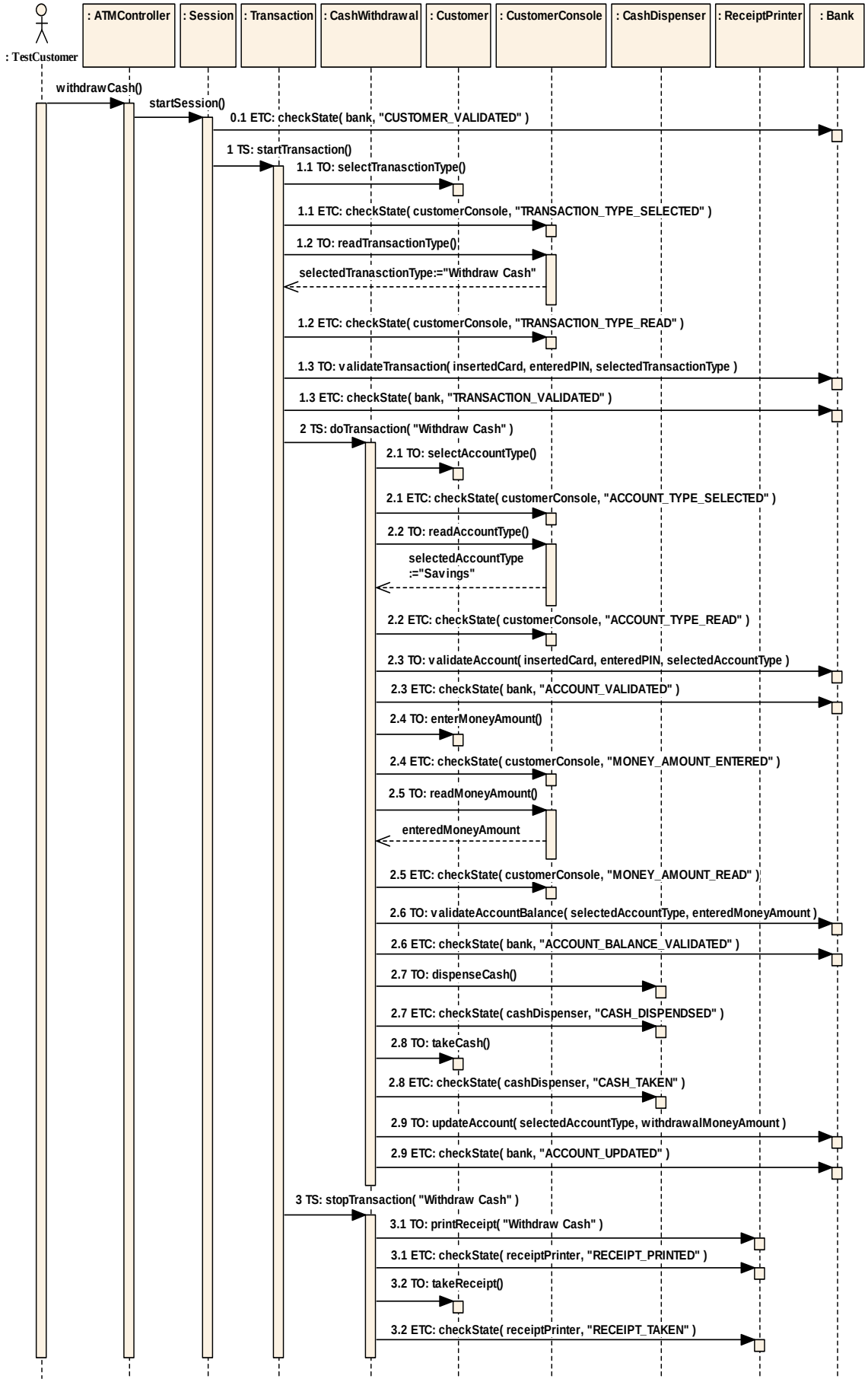


Figure C.7 Design Object Test Model: Design Test Sequence Diagram
(ATM TUC2 Core Test Scenario)

C.5 Designing and Generating Component Tests

For component test design and generation undertaken in the ATM case study, we apply the five main MBSCT methodological components: the integrated SCT process, the scenario-based CIT technique, the TbC technique, the TCR strategy and the CTM technique. This allows the ATM component test development to be model-based, process-based, scenario-based, contract-based, FDD-based and mapping-based (as described earlier in Chapter 4 to Chapter 8). Based on test models described in Section C.4, this section describes component test derivation undertaken in the ATM case study for the CIT purpose, and focuses on test sequence design (in Section C.5.1), component test design (in Section C.5.2), and component test generation (in Section C.5.3).

C.5.1 Test Sequence Design

For the CIT purpose, test sequence design is conducted in the ATM case study to organise and structure an array of logically-ordered relevant test artefacts (including test operations, test contracts and test elements) into test sequences for the ATM test scenarios captured with UML-based test models.

(1) ATM Session: test sequence design (as illustrated in Figure C.8)

Based on the corresponding four sub test scenarios (which are illustrated in Figure C.5), the test sequence designed for the ATM Session test scenario contains four (4) sub test sequences, with a total of eight (8) basic test groups. Each basic test group usually contains at least a pair of a test operation and its associated test contract. Sub test sequence #1 (i.e. TS: start Session) is a major sub test sequence and comprises five (5) basic test groups. Sub test sequence #3 (i.e. TS: continue another transaction) comprises one test group. Sub test sequence #4 (i.e. TS: stop Session) is a major sub test sequence and comprises two (2) basic test groups. Note that sub test sequence #2 (i.e. TS: perform Transaction) is further expanded and realised with the relevant ATM TUC test sequence (e.g. as illustrated in Figure C.9 and Figure C.10 below).

(2) ATM TUC1: test sequence design (as illustrated in Figure C.9)

Based on the corresponding three sub test scenarios (which are illustrated in Figure C.6), the test sequence designed for the ATM TUC1 test scenario contains three (3) sub test sequences, with a total of nine (9) basic test groups. Sub test sequence #1 (i.e. TS: start Transaction) comprises three (3) basic test groups. Sub test sequence #2 (i.e. TS: do current Transaction (“Inquire Balance”)) comprises four (4) basic test groups. Sub test sequence #3 (i.e. TS: stop current Transaction (“Inquire Balance”)) comprises two (2) basic test groups.

(3) ATM TUC2: test sequence design (as illustrated in Figure C.10)

Based on the corresponding three sub test scenarios (which are illustrated in Figure C.7), the test sequence designed for the ATM TUC2 test scenario contains three (3) sub test sequences with a total of fourteen (14) basic test groups. Sub test sequence #1 (i.e. TS: start Transaction) comprises three (3) basic test groups (as illustrated in Figure C.10 (a)). Sub test sequence #3 (i.e. TS: stop current Transaction ("Withdraw Cash")) comprises two (2) basic test groups (as illustrated also in Figure C.10 (a)). Sub test sequence #2 (i.e. TS: do current Transaction ("Withdraw Cash")), which is further expanded and illustrated in Figure C.10 (b), comprises nine (9) basic test groups.

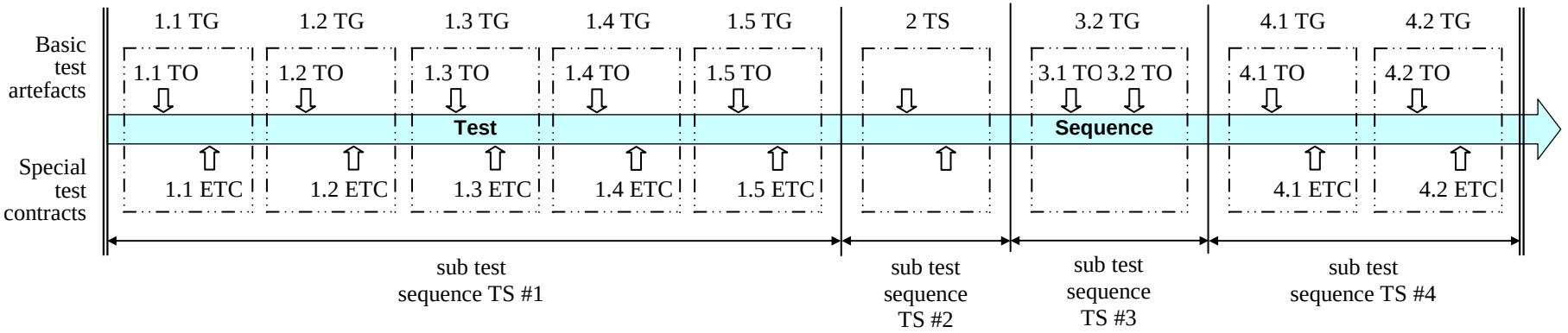


Figure C.8 Structured Test Sequence Design (ATM Session Test Scenario)

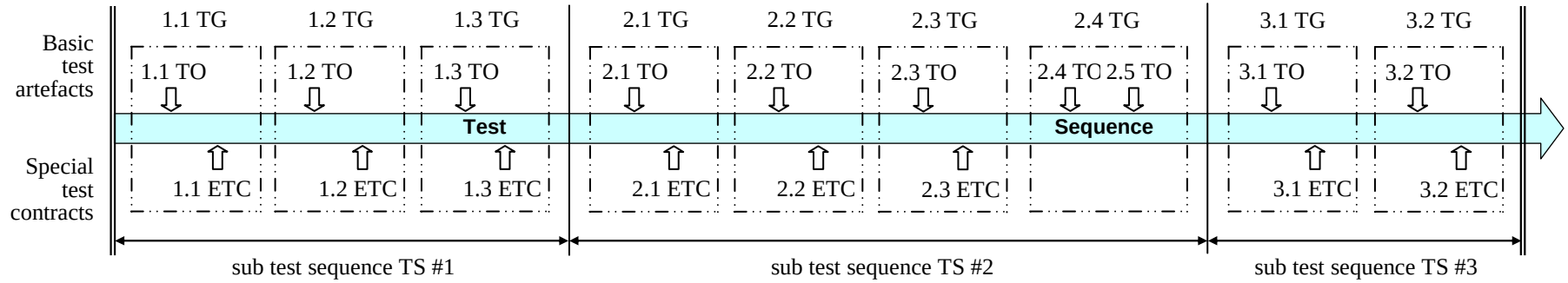
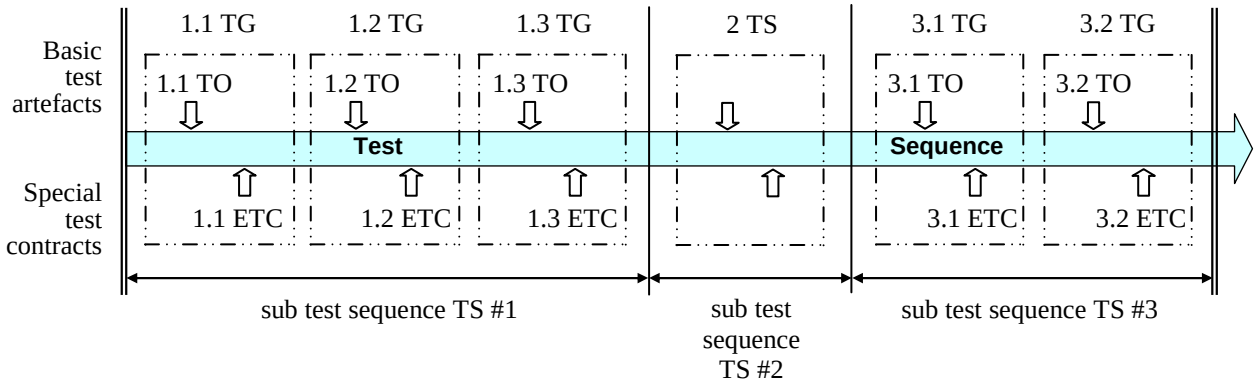
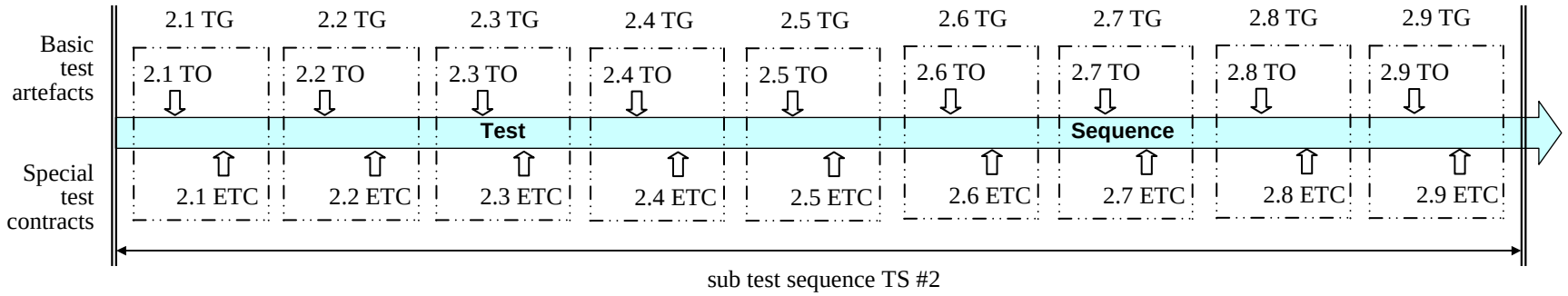


Figure C.9 Structured Test Sequence Design (ATM TUC1 Core Test Scenario)



(a) sub test sequence TS #1, TS #3



(b) sub test sequence TS #2

Figure C.10 Structured Test Sequence Design (ATM TUC2 Core Test Scenario)

C.5.2 Component Test Design

In the ATM case study, component test design is conducted to incorporate relevant test artefacts (including test sequences, test groups, test contracts and test operations with specified signatures, and test states) to design component tests in the corresponding integration test scenario for the CIT purpose.

(1) ATM Session: test design (as shown in Table C.2)

Table C.2 shows the component test design for the ATM Session test scenario, which illustrates relevant test artefacts and relationships for the CIT purpose. In this test design, there are a total of four (4) sub test scenarios/sequences, eight (8) test groups, nine (9) test operations, seven (7) test contracts, and seven (7) test states. Note that test contract ETC `checkState`(`cardReader`, "CARD_TAKEN") and associated test state "CARD_TAKEN" are the special test artefacts that are used as the overall preconditions/postconditions of the ATM Session test scenario.

(2) ATM TUC1: test design (as shown in Table C.3)

Table C.3 shows the component test design for the ATM TUC1 core test scenario, which illustrates relevant test artefacts and relationships for the CIT purpose. In this test design, there are a total of three (3) sub test scenarios/sequences, nine (9) test groups, ten (10) test operations, eight (8) test contracts, and eight (8) test states. Note that an initial test contract 0.1 ETC and associated test state "CUSTOMER_VALIDATED" are the special test artefacts that are used as the overall preconditions of the ATM TUC1 core test scenario.

(3) ATM TUC2: test design (as shown in Table C.4)

Table C.4 shows the component test design for the ATM TUC2 core test scenario, which illustrates relevant test artefacts and relationships for the CIT purpose. In this test design, there are a total of three (3) sub test scenarios/sequences, fourteen (14) test groups, fourteen (14) test operations, fourteen (14) test contracts, and fourteen (14) test states. Note that an initial test contract 0.1 ETC and associated test state "CUSTOMER_VALIDATED" are the special test artefacts that are used as the overall preconditions of the ATM TUC2 core test scenario.

Table C.2 Component Test Design (ATM Session Test Scenario): test sequences, test groups, test operations, test contracts and test states

Test Sequence	Test Group	Test Operation	Test Contract	Test State
perform Session		performSession()		
			0.1 ETC: checkState(cardReader, "CARD_TAKEN")	CARD_TAKEN
Sub Test Sequence #1 start Session		1 TS: startSession()		
	1.1 TG	1.1 TO: insertCard()	1.1 ETC: checkState(cardReader, "CARD_INSERTED")	CARD _INSERTED
	1.2 TG	1.2 TO: readCard()	1.2 ETC: checkState(cardReader, "CARD_READ")	CARD_READ
	1.3 TG	1.3 TO: enterPIN()	1.3 ETC: checkState(customerConsole, "PIN_ENTERED")	PIN_ENTERED
	1.4 TG	1.4 TO: readPIN()	1.4 ETC: checkState(customerConsole, "PIN_READ")	PIN_READ
	1.5 TG	1.5 TO: validateCustomer(insertedCard, enteredPIN)	1.5 ETC: checkState(bank, "CUSTOMER _VALIDATED")	CUSTOMER _VALIDATED
Sub Test Sequence #2 perform Transaction		2 TS: performTranasction()		
Sub Test Sequence #3 continue another transaction		3 TS: continueAnotherTransaction()		
	3.2 TG	3.1 TO: promptAnotherTransaction()		
		3.2 TO: indicateNoMoreTransaction()		
Sub Test Sequence #4 stop current Session		4 TS: stopSession()		
	4.1 TG	4.1 TO: ejectCard()	4.1 ETC: checkState(cardReader, "CARD_EJECTED")	CARD _EJECTED
	4.2 TG	4.2 TO: takeCard()	4.2 ETC: checkState(cardReader, "CARD_TAKEN")	CARD_TAKEN

Table C.3 Component Test Design (ATM TUC1 Core Test Scenario): test sequences, test groups, test operations, test contracts and test states

Test Sequence	Test Group	Test Operation	Test Contract	Test State
inquire balance		inquireBalance()		
			0.1 ETC: checkState(bank, "CUSTOMER _VALIDATED")	CUSTOMER _VALIDATED
Sub Test Sequence #1 start Transaction		1 TS: startTransaction()		
	1.1 TG	1.1 TO: selectTranasctionType()	1.1 ETC: checkState(customerConsole, "TRANSACTION _TYPE_SELECTED")	TRANSACTION _TYPE _SELECTED
	1.2 TG	1.2 TO: readTransactionType()	1.2 ETC: checkState(customerConsole, "TRANSACTION _TYPE_READ")	TRANSACTION _TYPE_READ
	1.3 TG	1.3 TO: validateTransaction(insertedCard, enteredPIN, selectedTransactionType)	1.3 ETC: checkState(bank, "TRANSACTION _VALIDATED")	TRANSACTION _VALIDATED
Sub Test Sequence #2 do current Transaction ("Inquire Balance")		2 TS: doTransaction("Inquire Balance")		
	2.1 TG	2.1 TO: selectAccountType()	2.1 ETC: checkState(customerConsole, "ACCOUNT _TYPE_SELECTED")	ACCOUNT _TYPE _SELECTED
	2.2 TG	2.2 TO: readAccountType()	2.2 ETC: checkState(customerConsole, "ACCOUNT _TYPE_READ")	ACCOUNT _TYPE _READ
	2.3 TG	2.3 TO: validateAccount(insertedCard, enteredPIN, selectedAccountType)	2.3 ETC: checkState(bank, "ACCOUNT _VALIDATED")	ACCOUNT _VALIDATED
	2.4 TG	2.4 TO: getAccountBalance()		
		2.5 TO: displayAccountBalance()		
Sub Test Sequence #3 stop current Transaction ("Inquire Balance")		3 TS: stopTransaction("Inquire Balance")		
	3.1 TG	3.1 TO: printReceipt("Inquire Balance")	3.1 ETC: checkState(receiptPrinter, "RECEIPT_PRINTED")	RECEIPT _PRINTED
	3.2 TG	3.2 TO: takeReceipt()	3.2 ETC: checkState(receiptPrinter, "RECEIPT_TAKEN")	RECEIPT _TAKEN

Table C.4 Component Test Design (ATM TUC2 Core Test Scenario): test sequences, test groups, test operations, test contracts and test states

Test Sequence	Test Group	Test Operation	Test Contract	Test State
withdraw cash		withdrawCash()		
			0.1 ETC: checkState(bank, "CUSTOMER_VALIDATED")	CUSTOMER_VALIDATED
Sub Test Sequence #2 start Transaction		1 TS: startTransaction()		
	1.1 TG	1.1 TO: selectTranasctionType()	1.1 ETC: checkState((customerConsole, "TRANSACTION_TYPE_SELECTED")	TRANSACTION_TYPE_SELECTED
	1.2 TG	1.2 TO: readTransactionType()	1.2 ETC: checkState(customerConsole, "TRANSACTION_TYPE_READ")	TRANSACTION_TYPE_READ
	1.3 TG	1.3 TO: validateTransaction(insertedCard, enteredPIN, selectedTransactionType)	1.3 ETC: checkState(bank, "TRANSACTION_VALIDATED")	TRANSACTION_VALIDATED
Sub Test Sequence #2 do current Transaction ("Withdraw Cash")		2 TS: doTransaction("Withdraw Cash")		
	2.1 TG	2.1 TO: selectAccountType()	2.1 ETC: checkState(customerConsole, "ACCOUNT_TYPE_SELECTED")	ACCOUNT_TYPE_SELECTED
	2.2 TG	2.2 TO: readAccountType()	2.2 ETC: checkState(customerConsole, "ACCOUNT_TYPE_READ")	ACCOUNT_TYPE_READ
	2.3 TG	2.3 TO: validateAccount(insertedCard, enteredPIN, selectedAccountType)	2.3 ETC: checkState(bank, "ACCOUNT_VALIDATED")	ACCOUNT_VALIDATED
	2.4 TG	2.4 TO: enterMoneyAmount()	2.4 ETC: checkState(customerConsole, "MONEY_AMOUNT_ENTERED")	MONEY_AMOUNT_ENTERED
	2.5 TG	2.5 TO: readMoneyAmount()	2.5 ETC: checkState(customerConsole, "MONEY_AMOUNT_READ")	MONEY_AMOUNT_READ
	2.6 TG	2.6 TO: validateAccountBalance(selectedAccountType, enteredMoneyAmount)	2.6 ETC: checkState(bank, "ACCOUNT_BALANCE_VALIDATED")	ACCOUNT_BALANCE_VALIDATED
	27 TG	2.7 TO: dispenseCash()	2.7 ETC: checkState(cashDispenser, "CASH_DISPENDSED")	CASH_DISPENDSED
	2.8 TG	2.8 TO: takeCash()	2.8 ETC: checkState(cashDispenser, "CASH_TAKEN")	CASH_TAKEN

	2.9 TG	2.9 TO: updateAccount(selectedAccountType, withdrawalMoneyAmount)	2.9 ETC: checkState(bank, "ACCOUNT_UPDATED")	ACCOUNT_UPDATED
Sub Test Sequence #3 stop current Transaction ("Withdraw Cash")		3 TS: stopTransaction("Withdraw Cash")		
	3.1 TG	3.1 TO: printReceipt("Withdraw Cash")	3.1 ETC: checkState(receiptPrinter, "RECEIPT_PRINTED")	RECEIPT_PRINTED
	3.2 TG	3.2 TO: takeReceipt()	3.2 ETC: checkState(receiptPrinter, "RECEIPT_TAKEN")	RECEIPT_TAKEN

C.5.3 Component Test Generation

This section presents the target CTS test case specifications that are derived in the ATM case study for the three selected ATM test scenarios as follows:

(1) The CTS test case specification for the ATM Session Test Design in the ATM Session test scenario – "Start Session" and "Stop Session" (as shown in Figure C.11)

Note that there are no specific test contracts associated with test operations in Test Set #3 (as shown in Figure C.11). These tests are related to the verification of the ATM's on-screen prompts/instructions and/or the customer's selections/responses to those prompts/instructions. Testing this aspect is not the focus in the current scope of the ATM case study.

```
... ... ... ...
<TestSpecification Name="ATM_Session_CTS.xml">
..<Desc>CTS test case specification for ATM Session: start/stop session</Desc>
... ... ... ...

..<TestSet Name="Session_TestSet_startSession">
....<Desc>Test Set #1: this test set examines Customer starts a new ATM session</Desc>

....<TestGroup Name="insertCard_groupedtests">
......<Desc>1.1 TG: grouped tests examine Customer inserts the ATM card into
           Card Reader</Desc>
......<TestOperation Name="insertCard_tests">
........<Desc>1.1 TO: examine setting Card Reader in the state of
             "CARD_INSERTED"</Desc>
........<TestMethod Name="insertCard" Target="customer">
..........<Desc>1.1 TO: set Card Reader in the state of "CARD_INSERTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.1 ETC: check Card Reader in the resulted correct state
               of "CARD_INSERTED"</Desc>
..........<Arg Name="aDevice" Source="cardReader" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="CARD_INSERTED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.1 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>
```

```
....<TestGroup Name="readCard_groupedtests">
......<Desc>1.2 TG: grouped tests examine ATM reads the inserted card
            from Card Reader</Desc>
......<TestOperation Name="readCard_tests">
........<Desc>1.2 TO: examine setting Card Reader in the state of "CARD_READ"</Desc>
........<TestMethod Name="readCard" Target="cardReader">
..........<Desc>1.2 TO: set Card Reader in the state of "CARD_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.2 ETC: check Card Reader in the resulted correct state
                of "CARD_READ"</Desc>
..........<Arg Name="aDevice" Source="cardReader" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="CARD_READ" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="enterPIN_groupedtests">
......<Desc>1.3 TG: grouped tests examine Customer enters the PIN from
            Customer Console (Keypad)</Desc>
......<TestOperation Name="enterPIN_tests">
........<Desc>1.3 TO: examine setting Customer Console (Keypad) in the state
              of "PIN_ENTERED"</Desc>
........<TestMethod Name="enterPIN" Target="customer">
..........<Desc>1.3 TO: set Customer Console (Keypad) in the state
                of "PIN_ENTERED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.3 ETC: check Customer Console (Keypad) in the resulted
                correct state of "PIN_ENTERED"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="PIN_ENTERED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="readPIN_groupedtests">
......<Desc>1.4 TG: grouped tests examine ATM reads the entered PIN
            from Customer Console (Keypad)</Desc>
......<TestOperation Name="readPIN_tests">
........<Desc>1.4 TO: examine setting Customer Console (Keypad) in the state
              of "PIN_READ"</Desc>
........<TestMethod Name="readPIN" Target="customerConsole">
..........<Desc>1.4 TO: set Customer Console (Keypad) in the state
                of "PIN_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.4 ETC: check Customer Console (Keypad) in the resulted
                correct state of "PIN_READ"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="PIN_READ" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.4 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="validatesCustomer_groupedtests">
......<Desc>1.5 TG: grouped tests examine Bank validates customer information</Desc>
......<TestOperation Name="validatesCustomer_tests">
........<Desc>1.5 TO: examine setting Bank in the state of "CUSTOMER_VALIDATED"</Desc>
........<TestMethod Name="validatesCustomer" Target="bank">
..........<Desc>1.5 TO: set Bank in the state of "CUSTOMER_VALIDATED"</Desc>
..........<Arg Name="insertedCard" Source="card" DataType="java.lang.Object" />
```

```
.........<Arg Name="enteredPIN" DataType="java.lang.Integer" />
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
.........<Desc>1.5 ETC: check Bank in the resulted correct state of
                 "CUSTOMER_VALIDATED"</Desc>
.........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
.........<Arg Name="aState" Source="CUSTOMER_VALIDATED"
               DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>1.5 ETC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="Session_TestSet_performTransaction">
....<Desc>Test Set #2: this test set examines performing an ATM transaction, and the
          related test spec is referred to the test spec of a specific ATM TUC</Desc>

..</TestSet>

..<TestSet Name="Session_TestSet_continueAnotherTransaction">
....<Desc>Test Set #3: this test set examines whether Customer is to
          do another transaction</Desc>

....<TestGroup Name="notDoAnotherTransaction_groupedtests">
......<Desc>3.2 TG: grouped tests examine Customer is not to
            do another transaction</Desc>
......<TestOperation Name="promptAnotherTransaction_tests">
........<Desc>3.1 TO: examine Customer Console on-screen prompts customer
              whether to do another transaction</Desc>
........<TestMethod Name="promptAnotherTransaction" Target="customerConsole">
.........<Desc>3.1 TO: Customer Console on-screen prompts customer
                 whether to do another transaction</Desc>
........</TestMethod>
......</TestOperation>
......<TestOperation Name="indicateNoMoreTransaction_tests">
........<Desc>3.2 TO: examine Customer indicates no more transaction</Desc>
........<TestMethod Name="indicateNoMoreTransaction" Target="customer">
.........<Desc>3.2 TO: Customer indicates no more transaction</Desc>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="Session_TestSet_stopSession">
....<Desc>Test Set #4: this test set examines Customer stops the current
          ATM session when indicating no more transaction</Desc>

....<TestGroup Name="ejectCard_groupedtests">
......<Desc>4.1 TG: grouped tests examine ATM ejects the inserted card
            from Card Reader</Desc>
......<TestOperation Name="ejectCard_tests">
........<Desc>4.1 TO: examine setting Card Reader in the state of
              "CARD_EJECTED"</Desc>
........<TestMethod Name="ejectCard" Target="cardReader">
.........<Desc>4.1 TO: set Card Reader in the state of "CARD_EJECTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
.........<Desc>4.1 ETC: check Card Reader in the resulted correct state
                 of "CARD_EJECTED"</Desc>
.........<Arg Name="aDevice" Source="cardReader" DataType="java.lang.Object" />
.........<Arg Name="aState" Source="CARD_EJECTED" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>4.1 ETC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>
```

```
....<TestGroup Name="takeCard_groupedtests">
......<Desc>4.2 TG: grouped tests examine Customer takes the ejected card
            from Card Reader</Desc>
......<TestOperation Name="takeCard_tests">
........<Desc>4.2 TO: examine setting Card Reader in the state of "CARD_TAKEN"</Desc>
........<TestMethod Name="takeCard" Target="customer">
..........<Desc>4.2 TO: set Card Reader in the state of "CARD_TAKEN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>4.2 ETC: check Card Reader in the resulted correct state
                of "CARD_TAKEN"</Desc>
..........<Arg Name="aDevice" Source="cardReader" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="CARD_TAKEN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>4.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure C.11 CTS Test Case Specification for the ATM Session Test Scenario

(2) The CTS test case specification for the ATM TUC1 Test Design in the ATM TUC1 core test scenario – “Inquire Balance” transaction (as shown in Figure C.12)

Note that there are no specific test contracts associated with test group 2.4 TG in Test Set #2 (as shown in Figure C.12). These tests are related to the examination of the numeric format representing dollars and cents that are displayed on the ATM Customer Console (Display/Screen). Testing this aspect is not the focus in the current scope of the ATM case study.

```
... ... ... ...
<TestSpecification Name="ATM_TUC1_CTS.xml">
..<Desc>CTS test case specification for ATM TUC1: Inquire Balance</Desc>
... ... ... ...

..<TestSet Name="TUC1_TestSet_startTransaction">
....<Desc>Test Set #1: this test set examines Customer starts the ATM
          transaction ("Inquire Balance")</Desc>

....<TestGroup Name="selectTranasctionType_groupedtests">
......<Desc>1.1 TG: grouped tests examine Customer selects the ATM transaction type
            ("Inquire Balance") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="selectTranasctionType_tests">
........<Desc>1.1 TO: examine setting Customer Console (Display/Screen) in
              the state of "TRANSACTION_TYPE_SELECTED" for the selected
              transaction type ("Inquire Balance")</Desc>
........<TestMethod Name="selectTranasctionType" Target="customer">
..........<Desc>1.1 TO: set Customer Console (Display/Screen) in the state
                of "TRANSACTION_TYPE_SELECTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.1 ETC: check Customer Console (Display/Screen) in the resulted
                correct state "TRANSACTION_TYPE_SELECTED"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
```

```
..........<Arg Name="aState" Source="TRANSACTION_TYPE_SELECTED"
               DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.1 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="readTransactionType_groupedtests">
......<Desc>1.2 TG: grouped tests examine ATM reads the selected transaction type
            ("Inquire Balance") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="readTransactionType_tests">
........<Desc>1.2 TO: examine setting Customer Console (Display/Screen) in
              the state of "TRANSACTION_TYPE_READ" for the read-in
              transaction type ("Inquire Balance")</Desc>
........<TestMethod Name="readTransactionType" Target="customerConsole">
..........<Desc>1.2 TO: set Customer Console (Display/Screen) in the state
                of "TRANSACTION_TYPE_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.2 ETC: check Customer Console (Display/Screen) in the resulted
                correct state of "TRANSACTION_TYPE_READ"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="TRANSACTION_TYPE_READ"
               DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="validateTransaction_groupedtests">
......<Desc>1.3 TG: grouped tests examine Bank validates the selected
            transaction type ("Inquire Balance")</Desc>
......<TestOperation Name="validateTransaction_tests">
........<Desc>1.3 TO: examine setting Bank in the state of "TRANSACTION_VALIDATED"
              for the selected transaction type ("Inquire Balance")</Desc>
........<TestMethod Name="validateTransaction" Target="bank">
..........<Desc>1.3 TO: set Bank in the state of "TRANSACTION_VALIDATED"</Desc>
..........<Arg Name="insertedCard" Source="card" DataType="java.lang.Object" />
..........<Arg Name="enteredPIN" DataType="java.lang.Integer" />
..........<Arg Name="selectedTransactionType" DataType="java.lang.String" />
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.3 ETC: check Bank in the resulted correct state of
                 "TRANSACTION_VALIDATED"</Desc>
..........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="TRANSACTION_VALIDATED"
               DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC1_TestSet_doTransaction("Inquire Balance")">
....<Desc>Test Set #2: this test set examines Customer does the current ATM
          transaction ("Inquire Balance")</Desc>

....<TestGroup Name="selectAccountType_groupedtests">
......<Desc>2.1 TG: grouped tests examine Customer selects the account type
            ("Savings") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="selectAccountType_tests">
........<Desc>2.1 TO: examine setting Customer Console (Display/Screen) in
              the state of "ACCOUNT_TYPE_SELECTED" for the selected
              account type ("Savings")</Desc>
........<TestMethod Name="selectAccountType" Target="customer">
```

```
..........<Desc>2.1 TO: set Customer Console (Display/Screen) in the state
                of "ACCOUNT_TYPE_SELECTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.1 ETC: check Customer Console (Display/Screen) in the resulted
                correct state "ACCOUNT_TYPE_SELECTED"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="ACCOUNT_TYPE_SELECTED"
              DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.1 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="readAccountType_groupedtests">
......<Desc>2.2 TG: grouped tests examine ATM reads the selected account type
            ("Savings") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="readAccountType_tests">
........<Desc>2.2 TO: examine setting Customer Console (Display/Screen) in
              the state of "ACCOUNT_TYPE_READ" for the read-in
              account type ("Savings")</Desc>
........<TestMethod Name="readAccountType" Target="customerConsole">
..........<Desc>2.2 TO: set Customer Console (Display/Screen) in the state
                of "ACCOUNT_TYPE_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.2 ETC: check Customer Console (Display/Screen) in the resulted
                correct state of "ACCOUNT_TYPE_READ"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="ACCOUNT_TYPE_READ" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="validateAccount_groupedtests">
......<Desc>2.3 TG: grouped tests examine Bank validates the selected
            account type ("Savings")</Desc>
......<TestOperation Name="validateAccount_tests">
........<Desc>2.3 TO: examine setting Bank in the state of "ACCOUNT_VALIDATED"
              for the selected account type ("Savings")</Desc>
........<TestMethod Name="validateAccount" Target="bank">
..........<Desc>2.3 TO: set Bank in the state of "ACCOUNT_VALIDATED"</Desc>
..........<Arg Name="insertedCard" Source="card" DataType="java.lang.Object" />
..........<Arg Name="enteredPIN" DataType="java.lang.Integer" />
..........<Arg Name="selectedAccountType" DataType="java.lang.String" />
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.3 ETC: check Bank in the resulted correct state of
                "ACCOUNT_VALIDATED"</Desc>
..........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="ACCOUNT_VALIDATED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="inquireBalance_groupedtests">
......<Desc>2.4 TG: grouped tests examine inquiring the available credit
            balance of the selected account ("Savings")</Desc>
......<TestOperation Name="getAccountBalance_tests">
........<Desc>2.4 TO: examine getting the available credit balance of
              the selected account ("Savings")</Desc>
........<TestMethod Name="getAccountBalance" Target="bank">
..........<Desc>2.4 TO: getting the available credit balance of
                the selected account ("Savings")</Desc>
```

```
........</TestMethod>
......</TestOperation>
......<TestOperation Name="displayAccountBalance_tests">
........<Desc>2.4 TO: examine Customer Console on-screen displays the available
              credit balance of the selected account ("Savings")</Desc>
........<TestMethod Name="displayAccountBalance" Target="customerConsole">
..........<Desc>2.4 TO: Customer Console on-screen displays the available
                credit balance of the selected account ("Savings")</Desc>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC1_TestSet_stopTransaction("Inquire Balance")">
....<Desc>Test Set #3: this test set examines Customer stops/finishes
          the current ATM transaction ("Inquire Balance")</Desc>

....<TestGroup Name="printReceipt_groupedtests">
......<Desc>3.1 TG: grouped tests examine ATM prints the receipt of the current ATM
            transaction ("Inquire Balance") from Receipt Printer</Desc>
......<TestOperation Name="printReceipt_tests">
........<Desc>3.1 TO: examine setting Receipt Printer in the state of
              "RECEIPT_PRINTED" for the current transaction ("Inquire Balance")</Desc>
........<TestMethod Name="printReceipt" Target="receiptPrinter">
..........<Desc>3.1 TO: set Receipt Printer in the state of "RECEIPT_PRINTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>3.1 ETC: check Receipt Printer in the resulted correct
                state of "RECEIPT_PRINTED"</Desc>
..........<Arg Name="aDevice" Source="receiptPrinter" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="RECEIPT_PRINTED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>3.1 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="takeReceipt_groupedtests">
......<Desc>3.2 TG: grouped tests examine Customer takes the printer receipt of the
            current ATM transaction ("Inquire Balance") from Receipt Printer</Desc>
......<TestOperation Name="takeReceipt_tests">
........<Desc>3.2 TO: examine setting Receipt Printer in the state of "RECEIPT_TAKEN"
              for the current ATM transaction ("Inquire Balance")</Desc>
........<TestMethod Name="takeReceipt" Target="customer">
..........<Desc>3.2 TO: set Receipt Printer in the state of "RECEIPT_TAKEN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>3.2 ETC: check Receipt Printer in the resulted correct
                state of "RECEIPT_TAKEN"</Desc>
..........<Arg Name="aDevice" Source="receiptPrinter" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="RECEIPT_TAKEN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>3.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure C.12 CTS Test Case Specification for the ATM TUC1 Core Test Scenario

(3) The CTS test case specification for the ATM TUC2 Test Design in the ATM TUC2 core test scenario– "Withdraw Cash" transaction (as shown in Figure C.13)

```
... ... ... ...
<TestSpecification Name="ATM_TUC2_CTS.xml">
..<Desc>CTS test case specification for ATM TUC2: Withdraw Cash</Desc>
... ... ... ...

..<TestSet Name="TUC2_TestSet_startTransaction">
....<Desc>Test Set #1: this test set examines starting the ATM
          transaction ("Withdraw Cash")</Desc>

....<TestGroup Name="selectTranasctionType_groupedtests">
......<Desc>1.1 TG: grouped tests examine Customer selects the ATM transaction type
            ("Withdraw Cash") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="selectTranasctionType_tests">
........<Desc>1.1 TO: examine setting Customer Console (Display/Screen) in
              the state of "TRANSACTION_TYPE_SELECTED" for the selected
              transaction type ("Withdraw Cash")</Desc>
........<TestMethod Name="selectTranasctionType" Target="customer">
..........<Desc>1.1 TO: set Customer Console (Display/Screen) in the state
                of "TRANSACTION_TYPE_SELECTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.1 ETC: check Customer Console (Display/Screen) in the resulted
                correct state "TRANSACTION_TYPE_SELECTED"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="TRANSACTION_TYPE_SELECTED"
                DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.1 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="readTransactionType_groupedtests">
......<Desc>1.2 TG: grouped tests examine ATM reads the selected transaction type
            ("Withdraw Cash") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="readTransactionType_tests">
........<Desc>1.2 TO: examine setting Customer Console (Display/Screen) in
              the state of "TRANSACTION_TYPE_READ" for the read-in
              transaction type ("Withdraw Cash")</Desc>
........<TestMethod Name="readTransactionType" Target="customerConsole">
..........<Desc>1.2 TO: set Customer Console (Display/Screen) in the state
                of "TRANSACTION_TYPE_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>1.2 ETC: check Customer Console (Display/Screen) in the resulted
                correct state of "TRANSACTION_TYPE_READ"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="TRANSACTION_TYPE_READ"
                DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>1.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="validateTransaction_groupedtests">
......<Desc>1.3 TG: grouped tests examine Bank validates the selected
            transaction type ("Withdraw Cash")</Desc>
......<TestOperation Name="validateTransaction_tests">
........<Desc>1.3 TO: examine setting Bank in the state of "TRANSACTION_VALIDATED"
              for the selected transaction type ("Withdraw Cash")</Desc>
........<TestMethod Name="validateTransaction" Target="bank">
..........<Desc>1.3 TO: set Bank in the state of "TRANSACTION_VALIDATED"</Desc>
..........<Arg Name="insertedCard" Source="card" DataType="java.lang.Object" />
```

```
.........<Arg Name="enteredPIN" DataType="java.lang.Integer" />
.........<Arg Name="selectedTransactionType" DataType="java.lang.String" />
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
.........<Desc>1.3 ETC: check Bank in the resulted correct state of
                "TRANSACTION_VALIDATED"</Desc>
.........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
.........<Arg Name="aState" Source="TRANSACTION_VALIDATED"
               DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>1.3 ETC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC1_TestSet_doTransaction("Withdraw Cash")">
....<Desc>Test Set #2: this test set examines Customer does the current ATM
          transaction ("Withdraw Cash")</Desc>

....<TestGroup Name="selectAccountType_groupedtests">
......<Desc>2.1 TG: grouped tests examine Customer selects the account type
            ("Savings") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="selectAccountType_tests">
........<Desc>2.1 TO: examine setting Customer Console (Display/Screen) in
              the state of "ACCOUNT_TYPE_SELECTED" for the selected
              account type ("Savings")</Desc>
........<TestMethod Name="selectAccountType" Target="customer">
.........<Desc>2.1 TO: set Customer Console (Display/Screen) in the state
                of "ACCOUNT_TYPE_SELECTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
.........<Desc>2.1 ETC: check Customer Console (Display/Screen) in the resulted
                correct state "ACCOUNT_TYPE_SELECTED"</Desc>
.........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
.........<Arg Name="aState" Source="ACCOUNT_TYPE_SELECTED"
                DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>2.1 ETC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="readAccountType_groupedtests">
......<Desc>2.2 TG: grouped tests examine ATM reads the selected account type
            ("Savings") from Customer Console (Display/Screen)</Desc>
......<TestOperation Name="readAccountType_tests">
........<Desc>2.2 TO: examine setting Customer Console (Display/Screen) in
              the state of "ACCOUNT_TYPE_READ" for the read-in
              account type ("Savings")</Desc>
........<TestMethod Name="readAccountType" Target="customerConsole">
.........<Desc>2.2 TO: set Customer Console (Display/Screen) in the state
                of "ACCOUNT_TYPE_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
.........<Desc>2.2 ETC: check Customer Console (Display/Screen) in the resulted
                correct state of "ACCOUNT_TYPE_READ"</Desc>
.........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
.........<Arg Name="aState" Source="ACCOUNT_TYPE_READ" DataType="java.lang.Object" />
.........<Result DataType="java.lang.Boolean" Save="y">
...........<Desc>2.2 ETC result: checkState must return true</Desc>
...........<Exp>true</Exp>
.........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="validateAccount_groupedtests">
......<Desc>2.3 TG: grouped tests examine Bank validates the selected
            account type ("Savings")</Desc>
```

```
......<TestOperation Name="validateAccount_tests">
........<Desc>2.3 TO: examine setting Bank in the state of "ACCOUNT_VALIDATED"
              for the selected account type ("Savings")</Desc>
........<TestMethod Name="validateAccount" Target="bank">
..........<Desc>2.3 TO: set Bank in the state of "ACCOUNT_VALIDATED"</Desc>
..........<Arg Name="insertedCard" Source="card" DataType="java.lang.Object" />
..........<Arg Name="enteredPIN" DataType="java.lang.Integer" />
..........<Arg Name="selectedAccountType" DataType="java.lang.String" />
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.3 ETC: check Bank in the resulted correct state of
                "ACCOUNT_VALIDATED"</Desc>
..........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="ACCOUNT_VALIDATED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.3 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="enterMoneyAmount_groupedtests">
......<Desc>2.4 TG: grouped tests examine Customer enters the withdrawal money
            amount from Customer Console (Keypad)</Desc>
......<TestOperation Name="enterMoneyAmount_tests">
........<Desc>2.4 TO: examine setting Customer Console (Keypad) in the state of
              "MONEY_AMOUNT_ENTERED" for the entered withdrawal money amount</Desc>
........<TestMethod Name="enterMoneyAmount" Target="customer">
..........<Desc>2.4 TO: set Customer Console (Keypad) in the state of
                "MONEY_AMOUNT_ENTERED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.4 ETC: check Customer Console (Keypad) in the resulted correct
                state of "MONEY_AMOUNT_ENTERED"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="MONEY_AMOUNT_ENTERED"
              DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.4 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="readMoneyAmount_groupedtests">
......<Desc>2.5 TG: grouped tests examine ATM reads the entered withdrawal money
            amount from Customer Console (Keypad)</Desc>
......<TestOperation Name="readMoneyAmount_tests">
........<Desc>2.5 TO: examine setting Customer Console (Keypad) in the state of
              "MONEY_AMOUNT_READ" for the read-in withdrawal money amount</Desc>
........<TestMethod Name="readMoneyAmount" Target="customerConsole">
..........<Desc>2.5 TO: set Customer Console (Keypad) in the state of
                "MONEY_AMOUNT_READ"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.5 ETC: check Customer Console (Keypad) in the resulted correct
                state of "MONEY_AMOUNT_READ"</Desc>
..........<Arg Name="aDevice" Source="customerConsole" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="MONEY_AMOUNT_READ" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.5 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="validateAccountBalance_groupedtests">
......<Desc>2.6 TG: grouped tests examine Bank validates the available credit
            balance of the selected account ("Savings") with the entered
            withdrawal money amount</Desc>
......<TestOperation Name="validateAccountBalance_tests">
........<Desc>2.6 TO: examine setting Bank in the state of
```

```
              "ACCOUNT_BALANCE_VALIDATED" for the selected account ("Savings")</Desc>
........<TestMethod Name="validateTransaction" Target="bank">
..........<Desc>2.6 TO: set Bank in the state of "ACCOUNT_BALANCE_VALIDATED"</Desc>
..........<Arg Name="selectedAccountType" DataType="java.lang.String" />
..........<Arg Name="enteredMoneyAmount" DataType="java.lang.Integer" />
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.6 ETC: check Bank in the resulted correct state of
                 "ACCOUNT_BALANCE_VALIDATED"</Desc>
..........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="ACCOUNT_BALANCE_VALIDATED"
                DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.6 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="dispenseCash_groupedtests">
......<Desc>2.7 TG: grouped tests examine ATM dispenses the withdrawal amount
            of cash notes from Cash Dispenser</Desc>
......<TestOperation Name="dispenseCash_tests">
........<Desc>2.7 TO: examine setting Cash Dispenser in the state
              of "CASH_DISPENDSED"</Desc>
........<TestMethod Name="dispenseCash" Target="cashDispenser">
..........<Desc>2.7 TO: set Cash Dispenser in the state of "CASH_DISPENDSED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.7 ETC: check Cash Dispenser in the resulted correct
                 state of "CASH_DISPENDSED"</Desc>
..........<Arg Name="aDevice" Source="cashDispenser" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="CASH_DISPENDSED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.7 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="takeCash_groupedtests">
......<Desc>2.8 TG: grouped tests examine Customer takes the dispensed
            cash notes from Cash Dispenser</Desc>
......<TestOperation Name="takeCash_tests">
........<Desc>2.8 TO: examine setting Cash Dispenser in the state
              of "CASH_TAKEN"</Desc>
........<TestMethod Name="takeCash" Target="customer">
..........<Desc>2.8 TO: set Cash Dispenser in the state of "CASH_TAKEN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.8 ETC: check Cash Dispenser in the resulted correct
                 state of "CASH_TAKEN"</Desc>
..........<Arg Name="aDevice" Source="cashDispenser" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="CASH_TAKEN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.8 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="updateAccount_groupedtests">
......<Desc>2.9 TG: grouped tests examine Bank updates the record of the selected
            account ("Savings") with the dispensed/withdrawn cash amount</Desc>
......<TestOperation Name="validateAccountBalance_tests">
........<Desc>2.9 TO: examine setting Bank in the state of
              "ACCOUNT_UPDATED" for the selected account ("Savings")</Desc>
........<TestMethod Name="updateAccount" Target="bank">
..........<Desc>2.9 TO: set Bank in the state of "ACCOUNT_UPDATED"</Desc>
..........<Arg Name="selectedAccountType" DataType="java.lang.String" />
..........<Arg Name="withdrawalMoneyAmount" DataType="java.lang.Integer" />
........</TestMethod>
```

```
........<TestMethod Name="checkState" Target="session">
..........<Desc>2.9 ETC: check Bank in the resulted correct state of
                "ACCOUNT_BALANCE_VALIDATED"</Desc>
..........<Arg Name="aBank" Source="bank" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="ACCOUNT_UPDATED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>2.9 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

..<TestSet Name="TUC1_TestSet_stopTransaction("Withdraw Cash")">
....<Desc>Test Set #3: this test set examines Customer stops/finishes
          the current ATM transaction ("Withdraw Cash")</Desc>

....<TestGroup Name="printReceipt_groupedtests">
......<Desc>3.1 TG: grouped tests examine ATM prints the receipt of the current ATM
            transaction ("Withdraw Cash") from Receipt Printer</Desc>
......<TestOperation Name="printReceipt_tests">
........<Desc>3.1 TO: examine setting Receipt Printer in the state of
              "RECEIPT_PRINTED" for the current transaction ("Withdraw Cash")</Desc>
........<TestMethod Name="printReceipt" Target="receiptPrinter">
..........<Desc>3.1 TO: set Receipt Printer in the state of "RECEIPT_PRINTED"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>3.1 ETC: check Receipt Printer in the resulted correct
                state of "RECEIPT_PRINTED"</Desc>
..........<Arg Name="aDevice" Source="receiptPrinter" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="RECEIPT_PRINTED" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>3.1 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

....<TestGroup Name="takeReceipt_groupedtests">
......<Desc>3.2 TG: grouped tests examine Customer takes the printer receipt of the
            current ATM transaction ("Withdraw Cash") from Receipt Printer</Desc>
......<TestOperation Name="takeReceipt_tests">
........<Desc>3.2 TO: examine setting Receipt Printer in the state of "RECEIPT_TAKEN"
              for the current ATM transaction ("Withdraw Cash")</Desc>
........<TestMethod Name="takeReceipt" Target="customer">
..........<Desc>3.2 TO: set Receipt Printer in the state of "RECEIPT_TAKEN"</Desc>
........</TestMethod>
........<TestMethod Name="checkState" Target="session">
..........<Desc>3.2 ETC: check Receipt Printer in the resulted correct
                state of "RECEIPT_TAKEN"</Desc>
..........<Arg Name="aDevice" Source="receiptPrinter" DataType="java.lang.Object" />
..........<Arg Name="aState" Source="RECEIPT_TAKEN" DataType="java.lang.Object" />
..........<Result DataType="java.lang.Boolean" Save="y">
............<Desc>3.2 ETC result: checkState must return true</Desc>
............<Exp>true</Exp>
..........</Result>
........</TestMethod>
......</TestOperation>
....</TestGroup>

..</TestSet>

... ... ... ...
</TestSpecification>
... ... ... ...
```

Figure C.13 CTS Test Case Specification for the ATM TUC2 Core Test Scenario

C.6 Evaluation Examples for Evaluating Adequate Test Artefact Coverage and Component Testability Improvement

In Chapter 9, Section 9.4.2 and Section 9.4.3 examines and evaluates the effectiveness of the MBSCT testing capabilities #4 and #5 (for adequate test artefact coverage and component testability improvement), particularly with the evaluation example #3 for the ATM special testing requirements #8 in the ATM case study. This section illustrates the other two evaluation examples #1 and #2 for the two ATM special testing requirements #3 and #7 (in Subsections C.6.1 and C.6.2 respectively).

C.6.1 Evaluation Example #1: Customer Validation

The ATM special testing requirement #3 (Customer Validation) is important in the ATM Session test scenario. Customer validation requires adequate test artefact coverage and testability for validating the customer eligibility for accessing the ATM system, that is, the customer must have a valid ATM card and PIN to correctly start an ATM session for accessing the ATM system.

Based on Section C.5 above and Section 9.4.2 in Chapter 9, the component test design for the ATM Session test scenario develops a special sub test sequence #1 that can exercise and examine all five testing-required control operations of the ATM card and PIN, including 1.1 TO, 1.2 TO, 1.3 TO, 1.4 TO and 1.5 TO. These test operations are adequate to bridge *Test-Gap #1* (as described in Section 5.2.4.2 in Chapter 5). In addition, the special sub test sequence #1 covers a set of appropriately-designed test contracts, including 1.1 ETC, 1.2 ETC, 1.3 ETC, 1.4 ETC and 1.5 ETC. These testing-support artefacts can adequately verify each of the five testing-required control operations for customer validation, which can bridge *Test-Gap #2* (as described in Section 5.2.4.2 in Chapter 5). Adequate testing artefact coverage improves component testability and enables testing to evaluate the relevant test results for customer validation. Therefore, the ATM component test design can improve component testability and fulfil the ATM special testing requirement #3: Customer Validation.

C.6.2 Evaluation Example #2: Account Selection Validation

The ATM special testing requirement #7 (Account Selection Validation) is important in the test scenario of each ATM TUC. Account selection validation requires that adequate test artefact

coverage and testability are needed to validate the customer-selected account access eligibility in the ATM system. In particular, the customer-selected account (e.g. “Savings” account) must be valid and must be linked to the inserted ATM card for performing the customer-selected ATM transaction.

Based on Section C.5 above and Section 9.4.2 in Chapter 9, the component test design for the ATM TUC1 core test scenario constructs a special sub test sequence #2 that can exercise and examine all three testing-required control operations of account selection, including 2.1 TO, 2.2 TO and 2.3 TO. These test operations are adequate and can bridge *Test-Gap #1* (as described in Section 5.2.4.2 in Chapter 5). Moreover, the special sub test sequence #2 contains an array of appropriately-designed test contracts, including 2.1 ETC, 2.2 ETC and 2.3 ETC. These testing-support artefacts can adequately verify each of the three testing-required control operations for account selection validation, which can bridge *Test-Gap #2* (as described in Section 5.2.4.2 in Chapter 5). Adequate testing artefact coverage enables testing to evaluate the relevant test results of account selection validation and thus improves component testability. Therefore, the ATM component test design can improve component testability and realise the ATM special testing requirement #7: Account Selection Validation.

C.7 Evaluation Examples for Fault Case Scenario Analysis and Fault Diagnostic Solution Design

In Chapter 9, Section 9.4.4.1 examines and evaluates the effectiveness of the MBSCT testing capabilities #3 and #6 for fault detection, diagnosis and localisation, by performing fault case scenario analysis and fault diagnostic solution design specifically with the evaluation example #3 for the ATM special testing requirements #8 in the ATM case study. For this FDD evaluation, this section shows two other evaluation examples #1 and #2 for the two ATM special testing requirements #3 and #7 (in Subsections C.7.1 and C.7.2 respectively).

C.7.1 Evaluation Example #1: Customer Validation

(1) Fault Case Scenario and Analysis

For the major fault/failure scenario of Customer Validation: The ATM/Bank system fails to validate the ATM-input customer information (e.g. card number and PIN), and/or fails to reject the customer’s access to the ATM while this validation is NOT fulfilled. The correct validation requires that the inserted-card number must be valid, the entered PIN must be valid, and the ATM-input customer information must be correct and identical to the customer information stored in the Bank system. A validation failure would allow the customer to access the ATM

while the customer-inserted card is invalid and/or the customer-entered PIN is invalid, which violates the ATM special testing requirement #3: Customer Validation.

(2) Fault-Related Test Scenario

This fault is covered by the ATM Session test scenario.

(3) Fault-Related ATM Device (or Fault-Related Bank Operation)

This fault is related to the Card Reader device, the Customer Console (Keypad) device, the Customer, and/or the Bank.

(4) Fault Diagnostic Solution

The fault diagnosis is CIT-related in the ATM Session test scenario. The fault diagnostic solution with the ATM Session test design must incorporate certain basic fault diagnostic solutions with the following related test groups (as described in Section C.5.2):

(a) Test group 1.1 TG comprises test operation 1.1 TO `insertCard`() and its associated test contract 1.1 ETC `checkState`(`cardReader`, "CARD_INSERTED") (as postcondition), and test state "CARD_INSERTED".

(b) Test group 1.2 TG comprises test operation 1.2 TO `readCard`() and its associated test contract 1.2 ETC `checkState`(`cardReader`, "CARD_READ") (as postcondition), and test state "CARD_READ".

(c) Test group 1.3 TG comprises test operation 1.3 TO `enterPIN`() and its associated test contract 1.3 ETC `checkState`(`customerConsole`, "PIN_ENTERED") (as postcondition), and test state "PIN_ENTERED".

(d) Test group 1.4 TG comprises test operation 1.4 TO `readPIN`() and its associated test contract 1.4 ETC `checkState`(`customerConsole`, "PIN_READ") (as postcondition), and test state "PIN_READ".

(e) Test group 1.5 TG comprises test operation 1.5 TO `validateCustomer`(`insertedCard`, `enteredPIN`) and its associated test contract 1.5 ETC `checkState`(`bank`, "CUSTOMER_VALIDATED") (as postcondition), and test state "CUSTOMER_VALIDATED".

C.7.2 Evaluation Example #2: Account Selection Validation

(1) Fault Case Scenario and Analysis

For the major fault/failure scenario of Account Selection Validation: The ATM/Bank system fails to validate the customer-selected account, and/or fails to reject the customer's access to the selected account while this validation is NOT fulfilled. The correct validation requires

that the customer-selected account must be valid for the customer's account in the Bank system, must be linked to the inserted ATM card, and can be accessed by the customer to perform the customer-selected ATM transaction. A validation failure would allow the customer to perform transactions on the selected account, which violates the ATM special testing requirement #7: Account Selection Validation.

(2) Fault-Related Test Scenario

This fault is covered by the test scenario of each ATM TUC, e.g. in the ATM TUC1 core test scenario.

(3) Fault-Related ATM Device (or Fault-Related Bank Operation)

This fault is related to the Customer Console (Display/Screen) device, the Customer, and/or the Bank.

(4) Fault Diagnostic Solution

The fault diagnosis is CIT-related in the ATM TUC1 core test scenario. The fault diagnostic solution with the ATM TUC1 test design must incorporate certain basic fault diagnostic solutions with the following related test groups (as described in Section C.5.2):

(a) Test group 2.1 TG comprises test operation 2.1 TO `selectAccountType`() and its associated test contract 2.1 ETC `checkState`(`customerConsole`, "ACCOUNT_TYPE_SELECTED") (as postcondition), and test state "ACCOUNT_TYPE_SELECTED".

(b) Test group 2.2 TG comprises test operation 2.2 TO `readAccountType`() and its associated test contract 2.2 ETC `checkState`(`customerConsole`, "ACCOUNT_TYPE_READ") (as postcondition), and test state "ACCOUNT_TYPE_READ".

(c) Test group 2.3 TG comprises test operation 2.3 TO `validateAccount`(`insertedCard`, `enteredPIN`, `selectedAccountType`) and its associated test contract 2.3 ETC `checkState`(`bank`, "ACCOUNT_VALIDATED") (as postcondition), and test state "ACCOUNT_VALIDATED".

C.8 Evaluation Examples for Evaluating Adequate Component Fault Coverage and Diagnostic Solutions and Results

In Chapter 9, Section 9.4.4.3 examines and evaluates the effectiveness of the MBSCT testing capability #6 for evaluating adequate component fault coverage and diagnostic solutions and

results, particularly with the evaluation example #3 for the ATM special testing requirements #8 in the ATM case study. For this FDD evaluation, this section presents two further evaluation examples #1 and #2 for the ATM special testing requirements #3 and #7 (in Subsections C.8.1 and C.8.2 respectively).

C.8.1 Evaluation Example #1: Customer Validation

This subsection evaluates the fault diagnostic solutions and results for diagnosing the possible faults that result in the same major requirement-violating fault FAULT_CUSTOMER against the ATM special testing requirement #3: Customer Validation. As described in Section C.7.1 and Table 9.7 in Chapter 9, we develop and apply the five individual basic fault diagnostic solutions in the ATM case study. Each basic fault diagnostic solution uses a basic test group to diagnose a directly/indirectly related fault in the ATM Session test scenario (as illustrated in Figure C.14).

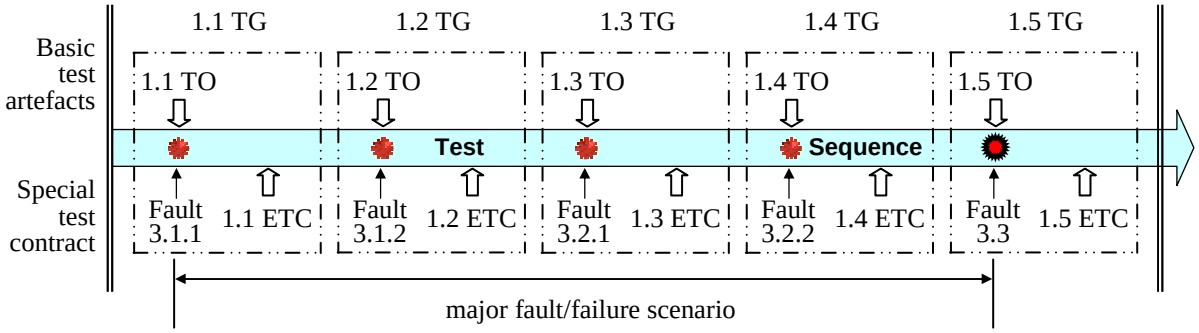


Figure C.14 Evaluation Example #1: Customer Validation
(Fault Diagnostic Solutions with the ATM Session Test Design)

The following describes the FDD evaluation for this major requirement-violating fault:

(1) Basic Fault 3.3 FAULT_CUSTOMER_VALIDATED (as shown in Table 9.7 in Chapter 9)

To diagnose the directly-related fault in the ATM Session test scenario, the ATM Session test design incorporates the first fault diagnostic solution that uses test group 1.5 TG to exercise test operation `1.5 TO validateCustomer( insertedCard, enteredPIN )`. This operation is verified by its associated test contract `1.5 ETC checkState( bank,` "CUSTOMER_VALIDATED") (as postcondition) and test state "CUSTOMER_VALIDATED".

If test contract 1.5 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `validateCustomer()` fails, causing the Bank system NOT to be in the correct control state of "CUSTOMER_VALIDATED" as expected. This means that the ATM/Bank system fails to validate the ATM-input customer infor-

mation (e.g. card number and PIN), and/or fails to reject the customer's access to the ATM while this validation is NOT fulfilled. In this fault case scenario, the ATM-input customer information is invalid in the Bank system, and the current customer is not permitted to access the ATM. This accords with the basic fault 3.3 FAULT_CUSTOMER_VALIDATED as described in Table 9.7, and the customer validation failure directly violates the ATM special testing requirement #3: Customer Validation.

Therefore, the basic fault 3.3 FAULT_CUSTOMER_VALIDATED is the directly-related fault that causes the major requirement-violating fault FAULT_CUSTOMER, which directly results in the major fault/failure scenario of Customer Validation as described in Section C.7.1. The first fault diagnostic solution is able to diagnose this directly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related Bank's operation `validateCustomer`().

(2) Basic Fault 3.1 FAULT_CARD (as shown in Table 9.7 in Chapter 9)

To diagnose possible indirectly-related faults that are associated with the ATM card in the ATM Session test scenario, the FDD evaluation further examines the following two fault case scenarios.

(2.1) Basic Fault 3.1.2 FAULT_CARD_READ (as shown in Table 9.7 in Chapter 9)

To diagnose an indirectly-related fault that is associated with the ATM card in the ATM Session test scenario, the ATM Session test design incorporates the second fault diagnostic solution that uses test group 1.2 TG to exercise test operation 1.2 TO `readCard`(). This operation is verified by its associated test contract 1.2 ETC `checkState`(`cardReader`, "CARD_READ") (as postcondition) and test state "CARD_READ".

If test contract 1.2 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the Card Reader device fails in the execution of operation `readCard`(), causing the Card Reader device NOT to be in the correct control state of "CARD_READ" as expected. This means that the ATM fails to read in the card information (e.g. card number) encoded on the customer-inserted ATM card, and/or the Card Reader device fails to eject the inserted but unreadable/unacceptable card. This accords with the basic fault 3.1.2 FAULT_CARD_READ as described in Table 9.7. The occurrence of this fault indicates a violated precondition, which causes the related succeeding operation `validateCustomer`() in the expected ATM Session test sequence NOT to be executed correctly, i.e. this validation operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Thus, the basic fault 3.1.2 FAULT_CARD_READ is an indirectly-related fault that causes

the directly-related fault 3.3 FAULT_CUSTOMER_VALIDATED, and then furthermore, as described in (1) above, indirectly results in the same major requirement-violating fault 3.3 FAULT_CUSTOMER. The second fault diagnostic solution is able to diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Card Reader device's operation `readCard`() can be corrected and removed.

(2.2) Basic Fault 3.1.1 FAULT_CARD_INSERTED (as shown in Table 9.7 in Chapter 9)

To diagnose an indirectly-related fault that is associated with the ATM card in the ATM Session test scenario, the ATM Session test design incorporates the third fault diagnostic solution that uses test group 1.1 TG to exercise test operation 1.1 TO `insertCard`(). This operation is verified by its associated test contract 1.1 ETC `checkState`(`cardReader`, "CARD_INSERTED") (as postcondition) and test state "CARD_INSERTED".

If test contract 1.1 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `insertCard`() fails, causing the Card Reader device NOT to be in the correct control state of "CARD_INSERTED" as expected. This means that the ATM card is inserted incorrectly by the customer into the card slot of the Card Reader device. While this fault occurs, the Card Reader device fails to eject the ATM card that is inserted incorrectly by the customer into the card slot, and/or the ATM fails to be ready for the customer to re-insert a card for a new ATM session. This accords with the basic fault 3.1.1 FAULT_CARD_INSERTED as described in Table 9.7. The occurrence of this fault indicates a violated precondition, which causes the succeeding operation `readCard`() in the expected ATM Session test sequence NOT to be executed correctly, i.e. this operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Hence, the basic fault 3.1.1 FAULT_CARD_INSERTED is an indirectly-related fault that causes the indirectly-related fault 3.1.2 FAULT_CARD_READ, and then indirectly results in the same major requirement-violating fault 3.3 FAULT_CUSTOMER. The third fault diagnostic solution is able to diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer and Card Reader device related operation `insertCard`() can be corrected and removed.

(3) Basic Fault 3.2 FAULT_PIN (as shown in Table 9.7 in Chapter 9)

To diagnose possible indirectly-related faults that are associated with the customer's PIN in the ATM Session test scenario, we need to further evaluate the following two fault case scenarios.

(3.1) Basic Fault 3.2.2 FAULT_PIN_READ (as shown in Table 9.7 in Chapter 9)

To diagnose an indirectly-related fault that is associated with the customer's PIN in the ATM Session test scenario, the ATM Session test design incorporates the fourth fault diagnostic solution that uses test group 1.4 TG to exercise test operation 1.4 TO `readPIN`(). This operation is verified by its associated test contract 1.4 ETC `checkState`(`customerConsole`, "PIN_READ") (as postcondition) and test state "PIN_READ".

If test contract 1.4 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the Customer Console (Keypad) device fails in the execution of operation `readPIN`(), causing the Customer Console (Keypad) device NOT to be in the correct control state of "PIN_READ" as expected. This means that the ATM fails to read in the customer's PIN entered from the Customer Console (Keypad) device, and/or fails to reject the entered but unreadable/unacceptable customer's PIN, and/or fails to allow the customer to re-enter a readable/acceptable customer's PIN (within the permitted three entries). This accords with the basic fault 3.2.2 FAULT_PIN_READ as described in Table 9.7. The occurrence of this fault indicates a violated precondition, which causes the related succeeding operation `validateCustomer`() in the expected ATM Session test sequence NOT to be executed correctly, i.e. this validation operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Thus, the basic fault 3.2.2 FAULT_PIN_READ is an indirectly-related fault that causes the directly-related fault 3.3 FAULT_CUSTOMER_VALIDATED, and then indirectly results in the same major requirement-violating fault 3.3 FAULT_CUSTOMER. The fourth fault diagnostic solution is able to diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer Console device's operation `readPIN`() can be corrected and removed.

(3.2) Basic Fault 3.2.1 FAULT_PIN_ENTERED (as shown in Table 9.7 in Chapter 9)

To diagnose an indirectly-related fault that is associated with the customer's PIN in the ATM Session test scenario, the ATM Session test design incorporates the fifth fault diagnostic solution that uses test group 1.3 TG to exercise test operation 1.3 TO `enterPIN`(). This operation is verified by its associated test contract 1.3 ETC `checkState`(`customerConsole`, "PIN_ENTERED") (as postcondition) and test state "PIN_ENTERED".

If test contract 1.3 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `enterPIN`() fails, causing the Customer Console (Keypad) device NOT to be in the correct control state of "PIN_ENTERED" as expected. This means that the customer's PIN is entered incorrectly by the customer from Customer Console (Keypad) device. While this fault occurs, the ATM fails to reject the customer's PIN that is entered incorrectly by the customer from the Customer Console (Keypad) device,

and/or fails to allow the customer to re-enter another PIN (within the permitted three entries). This accords with the basic fault 3.2.1 FAULT_PIN_ENTERED as described in Table 9.7. The occurrence of this fault indicates a violated precondition, which causes the succeeding operation `readPIN`() in the expected ATM Session test sequence NOT to be executed correctly, i.e. this operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Hence, the basic fault 3.2.1 FAULT_PIN_ENTERED is an indirectly-related fault that causes the indirectly-related fault 3.2.2 FAULT_PIN_READ, and then indirectly results in the same major requirement-violating fault 3.3 FAULT_CUSTOMER. The fifth fault diagnostic solution is able to diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer and Customer Console device related operation `enterPIN`() can be corrected and removed.

(4) Combined faults of the above five individual directly/indirectly related faults

Based on the FDD evaluation in (1) to (3) above (including (2.1) and (2.2), (3.1) and (3.2) above), a comprehensive fault diagnostic solution needs to incorporate the abovementioned five individual fault diagnostic solutions to detect and diagnose the combined faults of the above five individual directly/indirectly related faults against the same ATM special testing requirement #3: Customer Validation. The combined faults can be corrected and removed in the following fault-related operations:

(a) the Bank's operation `validateCustomer`(), and/or
(b) the Card Reader device's operation `readCard`(), and/or
(c) the Customer and Card Reader device related operation `insertCard`(), and/or
(d) the Customer Console device's operation `readPIN`(), and/or
(e) the Customer and Customer Console device related operation `enterPIN`().

C.8.2 Evaluation Example #2: Account Selection Validation

This subsection evaluates the fault diagnostic solutions and results for diagnosing the possible faults that result in the same major requirement-violating fault FAULT_ACCOUNT_SELECTION against the ATM special testing requirement #7: Account Selection Validation. As described in Section C.7.2 and Table 9.7 in Chapter 9, we develop and apply the three individual basic fault diagnostic solutions in the ATM case study. Each basic fault diagnostic solution uses a basic test group to diagnose a directly/indirectly related fault in the ATM TUC1 test scenario (as illustrated in Figure C.15).

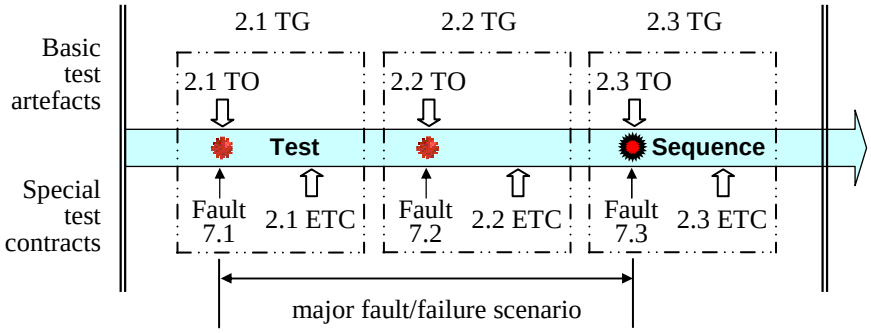


Figure C.15 Evaluation Example #2: Account Selection Validation (Fault Diagnostic Solutions with the ATM TUC1 Test Design)

The FDD evaluation for this major requirement-violating fault is described as follows:

(1) Basic Fault 7.3 FAULT_ACCOUNT_VALIDATED (as shown in Table 9.17 in Chapter 9)

To diagnose the directly-related fault in the ATM TUC1 test scenario, the ATM TUC1 test design incorporates the first fault diagnostic solution that uses test group 2.3 TG to exercise test operation 2.3 TO `validateAccount( insertedCard, enteredPIN, selectedAccountType )`; this operation is verified by its associated test contract 2.3 ETC `checkState( bank`, "ACCOUNT_VALIDATED") (as postcondition) and test state "ACCOUNT_VALIDATED".

If test contract 2.3 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `validateAccount`() fails, causing the Bank system NOT to be in the correct control state of "ACCOUNT_VALIDATED" as expected. This means that the ATM/Bank system fails to validate the customer-selected account, and/or fails to reject the customer's access to the selected account while this validation is NOT fulfilled. In this fault case scenario, the customer-selected account is invalid and/or inaccessible in the Bank system, and the current customer is not permitted to access the customer-selected account for doing any ATM transaction. This accords with the basic fault 7.3 FAULT_ACCOUNT_VALIDATED as described in Table 9.7, and the account validation failure directly violates the ATM special testing requirement #7: Account Selection Validation.

Therefore, the basic fault 7.3 FAULT_ACCOUNT_VALIDATED is the directly-related fault that causes the major requirement-violating fault FAULT_ACCOUNT_SELECTION, which directly results in the major fault/failure scenario of Account Selection Validation as described in Section C.7.2. The first fault diagnostic solution is able to diagnose this directly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault can be corrected and removed in the fault-related Bank's operation `validateAccount`().

(2) Basic Fault 7.2 FAULT_ACCOUNT_TYPE_READ (as shown in Table 9.7 in Chapter 9)

To diagnose an indirectly-related fault in the ATM TUC1 test scenario, the ATM TUC1 test design incorporates the second fault diagnostic solution that uses test group 2.2 TG to exercise test operation 2.2 TO `readAccountType`(); this operation is verified by its associated test contract 2.2 ETC `checkState`(`customerConsole`, "ACCOUNT_TYPE_READ") (as postcondition) and test state "ACCOUNT_TYPE_READ".

If test contract 2.2 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the Customer Console (Display/Screen) device fails in the execution of operation `readAccountType`(), causing the Customer Console (Display/Screen) device NOT to be in the correct control state of "ACCOUNT_TYPE_READ" as expected. This means that the ATM fails to read in the account type selected from the Customer Console (Display/Screen) device, and/or fails to reject the selected but unreadable/unacceptable account type, and/or fails to allow the customer to re-select a readable/acceptable account. This accords with the basic fault 7.2 FAULT_ACCOUNT_TYPE_READ as described in Table 9.7. The occurrence of this fault indicates a violated precondition, which causes the related succeeding operation `validateAccount`() in the expected ATM TUC1 test sequence NOT to be executed correctly, i.e. this validation operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Thus, the basic fault 7.2 FAULT_ACCOUNT_TYPE_READ is an indirectly-related fault that causes the directly-related fault 7.3 FAULT_ACCOUNT_VALIDATED, and then indirectly results in the same major requirement-violating fault FAULT_ACCOUNT_SELECTION. The second fault diagnostic solution is able to diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer Console device's operation `readAccountType`() can be corrected and removed.

(3) Basic Fault 7.1 FAULT_ACCOUNT_TYPE_SELECTED (as shown in Table 9.7 in Chapter 9)

To diagnose an indirectly-related fault in the ATM TUC1 test scenario, the ATM TUC1 test design incorporates the third fault diagnostic solution that uses test group 2.1 TG to exercise test operation 2.1 TO `selectAccountType`(); this operation is verified by its associated test contract 2.1 ETC `checkState`(`customerConsole`, "ACCOUNT_TYPE_SELECTED") (as postcondition) and test state "ACCOUNT_TYPE_SELECTED".

If test contract 2.1 ETC returns *false*, this fault diagnostic solution has detected and diagnosed the following fault: the execution of operation `selectAccountType`() fails,

causing the Customer Console (Display/Screen) device NOT to be in the correct control state of "ACCOUNT_TYPE_SELECTED" as expected. This means that the type of bank account is selected incorrectly by the customer from the Customer Console (Display/Screen) device. While this fault occurs, the ATM fails to reject the account type that is selected incorrectly by the customer from the Customer Console (Display/Screen) device, and/or fails to allow the customer to re-select another bank account. This accords with the basic fault 7.1 FAULT_ACCOUNT_TYPE_SELECTED as described in Table 9.7. The occurrence of this fault indicates a violated precondition, which causes the succeeding operation `readAccountType`() in the expected ATM TUC1 test sequence NOT to be executed correctly, i.e. this operation cannot be executed as expected or its execution fails in the expected operation execution sequence.

Hence, the basic fault 7.1 FAULT_ACCOUNT_TYPE_SELECTED is an indirectly-related fault that causes the indirectly-related fault 7.2 FAULT_ACCOUNT_TYPE_READ, and then indirectly results in the same major requirement-violating fault FAULT_ACCOUNT_SELECTION. The third fault diagnostic solution is able to diagnose this indirectly-related fault. Following the CBFDD guidelines (as described earlier in Section 7.5.5), the diagnosed fault that is associated with the Customer and Customer Console device related operation `selectAccountType`() can be corrected and removed.

(4) Combined faults of the above three individual directly/indirectly related faults

Based on the FDD evaluation in (1) to (3) above, a comprehensive fault diagnostic solution needs to incorporate the abovementioned three individual fault diagnostic solutions to detect and diagnose the combined faults of the above three individual directly/indirectly related faults against the same ATM special testing requirement #7: Account Selection Validation. The combined faults can be corrected and removed in the following fault-related operations:

(a) the Bank's operation `validateAccount`(), and/or

(b) the Customer Console device's operation `readAccountType`(), and/or

(c) the Customer and Customer Console device related operation `selectAccountType`().