

PhD Confirmation Report

Model-Driven Framework for Context-Dependent Testing of Components

Candidate:

Abu Zafer Javed

Advisory Team:

Dr Paul Anthony Strooper

Dr Geoffrey Norman Watson

August 2007

School of Information Technology and Electrical Engineering

The University of Queensland

This page is left blank intentionally

Table of Contents

Abstract	5
1 Introduction	6
1.1. Model-driven architecture	6
1.2. Component-based software development.....	7
1.3. Context-dependent component testing	7
1.4. The problems in context-dependent testing of components	8
1.5. A model-driven framework for context-dependent testing of components.....	9
1.6. Contributions	11
1.7. Document overview.....	11
2 Related Work	12
2.1 Model-based testing.....	12
2.2 Descriptions of use case charts.....	14
2.3 Model-driven testing	15
2.4 Component-based testing	16
2.4.1 Built-in testing	16
2.4.2 Component metadata	17
2.4.3 Component certification	18
2.4.4 Testable architecture.....	20
2.4.5 Discussion.....	21
3 Work Done: Generating unit test cases using model-driven architecture	22
3.1 Generating Test Cases using MDA	23
3.1.1 Step 1: Transforming SMC into xUnit	25
3.1.2 Step 2: Generating JUnit from xUnit.....	25
3.1.3 SMC Meta-model	26
3.1.4 xUnit Meta-model	27
3.2 Tracing.....	28
3.3 An Example of Test Case Generation	28
3.3.1 An Example Sequence Diagram.....	29
3.3.2 SMC Model	31
3.3.3 xUnit Model.....	31
3.3.4 Test Data File	32
3.3.5 JUnit Test Case.....	33
3.3.6 Traces.....	35
3.4 Discussion.....	36
4 Research Plan.....	39
4.1 The model-driven framework for context-dependent testing of components	39
4.2 Contributions	43
4.3 Tasks.....	44
4.4 Timeline.....	45
References.....	46

List of Figures

Figure 1: Component testing vs context-dependent component testing.....	8
Figure 2: Comparison of CB testing approaches.....	21
Figure 3: Overall Process	23
Figure 4: Overview of Methodology	24
Figure 5: An Example Tefkat Rule.....	25
Figure 6: Example MOFScript Rules	26
Figure 7: SMC Meta-model.....	27
Figure 8: xUnit Meta-model	28
Figure 9: An Example Sequence Diagram	30
Figure 10: SMC Model.....	32
Figure 11: xUnit Model	32
Figure 12: Test Data File	33
Figure 13: JUnit Test Case	34
Figure 14: Execution Trace	35
Figure 15: Code Reusability Matrix	36
Figure 16: Rules for mapMethod, mapParameter and mapExpectedValue	38
Figure 17: Adequacy of CDCT testing and component acceptability.....	40
Figure 18: Adequacy of CDCT and CT	41

Acronyms

CB Testing	Component-Based Testing
CB Software	Component-Based Software
CDCT	Context-Dependent Component Testing
CIM	Computation Independent Model
COTS	Commercial-Off-The-Shelf
CSD	Component-based Software Development
CT	Component Testing
MDA	Model-Driven Architecture
MTCG	Model Driven Test Case Generator
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
SMC	Sequence of Method Calls
UCC	Use Case Chart

Abstract

The aim of this project is to devise a novel model-driven framework for context-dependent testing of components (CDCT). Model-driven architecture (MDA) is an emerging approach for developing software applications from high-level models. In MDA, models at various levels of abstraction are used to automate software development and testing activities. Component-based Software Development (CSD) is a process in which software applications are developed by reusing existing components. This project will investigate the use of MDA for testing software applications that are developed using CSD.

When a general-purpose component is used in a new context, it needs to be thoroughly tested for that context. We propose to model the usage scenarios of a component, in a context of interest, using use cases and interaction diagrams. From these scenarios, test cases for performing CDCT are automatically generated. We then evaluate and extend test adequacy of the generated test cases by comparing them with the test cases that were executed during component testing by the component developer, and attached to the component as component metadata. Finally, we execute the more adequate set of test cases to test the component for the new context.

This approach is novel in that it will apply the emerging MDA technology to context-dependent testing of components. The proposed framework may benefit from the advantages of an MDA-based approach, such as portability, interoperability and maintainability. Another novelty is the use of component metadata to evaluate and extend the adequacy of CDCT. We will develop a prototype tool for the proposed framework and evaluate the framework and tool support.

1 Introduction

The aim of this project is to devise a novel model-driven framework for context-dependent testing of components.

1.1. Model-driven architecture

Model-Driven Architecture (MDA) is an initiative by the Object Management Group (OMG) to support the development of interoperable, portable and reusable software systems [1]. In MDA, models at various levels of abstraction are the central software design artifacts. They are used to facilitate both abstraction and automated development. MDA can contribute throughout the software development life cycle. Business analysts can develop a business model which is a computation independent model (CIM). Architects and designers develop Platform Independent Models (PIM) using the CIM. Developers and testers derive Platform Specific Models (PSM) from the PIM, for generating application code and test code respectively. Software maintenance teams can apply model slicing techniques to identify parts of the model that relate to a functionality that is being changed. Some of the advantages of an MDA-based approach are:

1. MDA tools can partially automate the development process by generating most of the code from models, resulting in less code to hand-craft [2].
2. MDA tools for reverse engineering can automate synthesis of software models [3].
3. MDA tools can help in generating test cases from software models to verify the software implementation [4].

A simple use of MDA is to model a system in a platform-independent modelling language (e.g. UML). The PIM can then be transformed into a PSM by executing transformation specifications that are mappings between the PIM and some implementation language (e.g. Java) [5]. The same process of transforming PIM to PSM can be used for automating the generation of test cases (e.g. JUnit [6] test case for Java).

1.2. Component-based software development

Component-based Software Development (CSD) is a process in which software applications are developed by reusing readily available components [7]. Two important advantages of CSD are shorter development time and lower development cost [8].

The term “component” has many definitions. We adopt Szyperski’s definition, who defines a component as “a unit of composition with contractually specified interfaces and explicit context dependencies” [9]. In CSD, the component provider develops component(s) and a set of interfaces to the component(s). The component user reuses the component(s), using their interfaces, to develop software applications. The interface of a component is a collection of service access points along with their semantic specifications [10]. These interfaces are the methods, implemented by the component developer, through which the reusing system interacts with the component. Software developed using this process is called Component-Based (CB) software.

1.3. Context-dependent component testing

Although CSD has various advantages over traditional software development, it complicates software testing. The component developers perform component testing to confirm that components exhibit the desired behaviour. Component testing refers to all activities that are related to testing of a component at the time of its development. This is often referred to as unit or module testing [11], and it increases the reliability of the component in CB software. However, software testing can show the presence of defects, but not their absence [12]. The component developer’s testing may fail to test the component sufficiently because:

- i) The use of the component is novel and may not be tested at all.
- ii) Components are generally developed to provide a wide range of functionality to achieve greater applicability [13]. However, the component usage (in the novel context) may be intensive over part of a component’s range of use, which was uniformly (averagely) tested (because the component developer also tested functionality that will not be used in the new context).

The component user would benefit from more intensive testing over the context that applies to the component. This context-dependent component testing (CDCT), which is also known as CB testing, refers to all activities that are related to testing in the scope of

a CSD [14]. CDCT testing aims to increase the reliability of the component for the new context. CDCT allows the component user to focus on finding errors related to the use of the component in a specific context. Figure 1 shows the relationship between component testing, which is uniform across different possible uses, and CDCT, which is more thorough for a particular use.

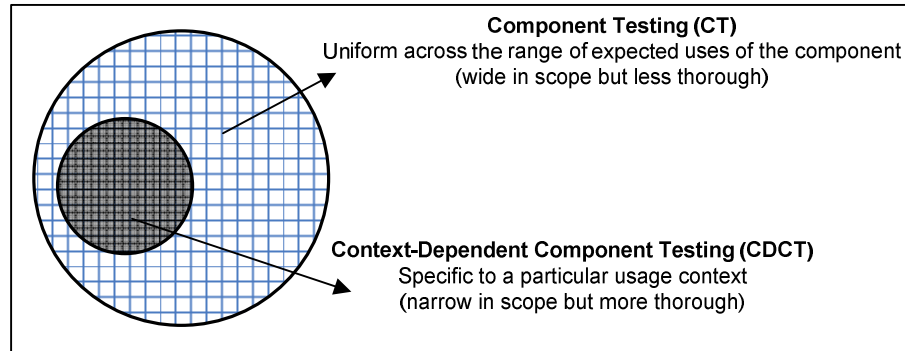


Figure 1: Component testing vs context-dependent component testing

The significance of context in component testing is demonstrated by the failure of the ARIANE 5 rocket which was launched in 1996 [15]. It crashed 40 seconds after take-off. The investigation revealed that a component of the ARIANE 5 that was taken from the ARIANE 4 failed to function properly. The reason for the failure was the higher initial velocity of the ARIANE 5 compared to the ARIANE 4. The reused component worked fine in the ARIANE 4 but it failed with the ARIANE 5. However, more thorough testing of the component in the context of the ARIANE 5 (CDCT) might have detected the problem.

An example of a slightly different nature is the reuse of a component on a different platform (operating system or machine architecture), which is different from the platform used for the component development and testing. These platforms may have a different way of data processing, which allows for different interpretations of the information exchanged between the CB software and the component [16].

1.4. The problems in context-dependent testing of components

CDCT has its own challenges and limitations. The most important problems are [17]:

- i) Unexpected behaviour of component(s) when reused in a new context
- ii) Lack of access to internal working of a component
- iii) Test adequacy criteria for component reuse

The most significant challenge in CSD is the testing of a component in a new context [14]. There is a possibility that a component is developed and tested in one context (by the component provider) and used in another context (by the component user) for which it was not tested. The reusing system can use the component in such a way that different states are activated and different paths are exercised. Therefore, the use of a component in a new context can expose previously undiscovered errors. Testing of a component in the context in which it is being reused is necessary as it can discover the defects that are specific to this context. Moreover, a more thorough testing can be performed by focusing on the particular usage of the component.

The second problem is the lack of access to the internal working of the component. For some components, such as Commercial-Off-The-Shelf (COTS) components, source code and design artefacts may not be available to the component user. This lack of information limits observability of a component. Observability is the extent to which inputs, outputs and behaviour of a component can be observed [18]. The lack of observability affects component testability [19]. It reduces the confidence that testing yields because, simply by examining outputs of a component, we cannot guarantee that it is working correctly. The correct output is not enough to decide on the correct working of a component because sometimes a component behaves incorrectly and still returns the correct output value. This incorrect behaviour of a component can affect the behaviour of the CB software.

Criteria for adequate testing of a component at the time of its reuse is another concern [20]. Component interfaces can provide some information about the component model, but they do not provide enough information for devising test adequacy criteria [13]. The component user often picks test adequacy criteria like executing all method calls that are part of the component's interface, but this may not be sufficient [21].

1.5. A model-driven framework for context-dependent testing of components

We propose a novel, model-driven framework for context-dependent testing of components to address the first and the third problems discussed in Section 1.4, which are the testing of a component in a new context and determining test adequacy criteria for component reuse. The proposed framework consists of the following steps.

i) Modelling component usage scenarios and generating test cases for CDCT

First, we model the actual component usage, required by the component user, in the CB software using a use case chart (UCC) [22]. From the interaction diagrams, which are part of the use case chart, we generate test cases for performing CDCT. We generate test cases only for those use cases in which the component is used. The test cases consist of a sequence of method calls (SMC).

ii) Evaluating and extending test adequacy of CDCT

We then evaluate the adequacy of the test cases, generated for CDCT, using the component-metadata provided as developer certification. Developer certification is a component testing (CT) technique in which the component developer executes test cases, and attaches these test cases to the component, as component metadata [23]. We determine and extend the adequacy of CDCT by comparing the test cases, generated for CDCT, with the test cases that were executed during developer certification.

CDCT is considered to be adequate if it includes (executes) all the test cases that were executed during the component certification. If there are some test cases that were executed during component certification and that are not included in CDCT, we review those test cases to decide if they can increase the adequacy of CDCT. If those test cases relate to component functionality that is used in the new context, we add them to the test cases that we generated for CDCT.

The test cases that are generated for CDCT but that were not part of component certification, indicate a weakness in component testing, i.e., the component was not adequately tested for the new context. As these test cases are devised at system-level, each of them indicates a functionality of the component that was not tested before. Therefore, we review these test cases to decide if they should be expanded (increased in number) to cover the functionality of the component that was not tested during CT.

Our approach for evaluating the adequacy of CDCT is based on comparison of test cases and test suites. Comparison of test cases is a challenging task because test cases that are not syntactically identical may still test the same functionality and should therefore be considered equivalent. Similarly, test suites can be compared using different techniques that have their own advantages or issues such as simplicity and scalability. We shall define different criteria for comparing test cases, and approaches

for comparing test suites. We shall evaluate these criteria and approaches to decide which is the most suitable for our purposes.

iii) Executing the test cases to test the component for a new context

We transform the test cases, generated for CDCT, into concrete and executable test cases using a model-driven approach. Finally, we execute these concrete test cases to test the component for the new context.

1.6. Contributions

This research project will make the following contributions.

- i) A novel model-driven framework for context-dependent testing of components
- ii) A novel algorithm for generating test case sequences from use case charts
- iii) A novel method for evaluating and extending test adequacy of CDCT
- iv) Criteria for comparing test cases and test suites
- v) Prototype tool support for the framework
- vi) Evaluation of the framework and tool support

1.7. Document overview

This document is organised into four sections. Section 1 provides an introduction to the project and the report. Section 2 surveys related work. Section 3 describes the work done so far. Section 4 outlines the research plan.

2 Related Work

Software testing is an important activity of a software development process, which is aimed at assuring the quality of the product. Software applications are tested to ensure their correct working. Myers defines software testing as a “process of trying to discover every conceivable fault or weakness in a work product” [24]. Traditionally, software products have been tested based upon their specifications or implementations [25]. Recently, model-based testing has become popular [26-28]. Model-based testing refers to processes, methods or technologies that use software models to perform testing activities. One reason for its popularity is that software models can contain both static and behavioural information, which provides a sound base for conducting testing activities [29]. More recently, model-driven testing has started to emerge. Model-driven testing is a form of model-based testing that is based on models, meta-models and transformation rules [30]. As our research focuses on model-driven testing of component-based software using use case charts and interaction diagrams, we will review related work in the following areas:

- i) Model-based testing in general, and testing using interaction diagrams in particular
- ii) Descriptions of use case charts
- iii) Model-driven testing
- iv) Component-based testing

2.1 Model-based testing

Unified Modelling Language (UML) is a de facto industry standard for modelling software applications [31]. Researchers are investigating different types of UML diagrams to support software testing activities. The UML diagrams that are widely used to automate software testing are: use case, activity, state machine and interaction diagrams. Use cases represent a specific use of a system. They are used for automatic generation of test cases for implementation verification [32, 33]. Activity diagrams are an object-oriented equivalent of Flow Charts and Data Flow Diagrams. They are used for validating software workflows [34] and generating test cases for implementation verification [35-37]. State machines represent the state-dependent behaviour of a

system. They are used to validate software models [38-40] and to verify software implementation [41-45].

UML interaction diagrams [31] represent dynamic interactions among objects, components or sub-systems. They consist of collaboration diagrams and sequence diagrams. The former emphasise the organisational structure of interacting components, and the latter show interactions in a time sequence manner. Interaction diagrams are used to determine test requirements, generate test data, and verify software implementation.

Abdurazik and Offutt [46] describe the derivation of test requirements criteria for both static and dynamic testing from collaboration diagrams. Supavita and Suwannasart [47] discuss test requirements for polymorphic interactions in UML sequence diagrams. Andrews et al. [48] propose test adequacy criteria from sequence diagrams that are defined in terms of condition coverage, full predicate coverage, each message on a link, and all message paths. Briand and Labiche [49] propose the TOTEM (Testing Objectoriented systems with the unified Modelling language) methodology to derive system test requirements by analysing UML artifacts such as class, use case, interaction diagrams and OCL constraints across these artifacts.

Samuel et al. [50] propose an algorithm for generating test data from UML sequence diagrams. In this technique, dynamic slices are generated at every message point in the sequence diagram and then the test cases are generated for each slice. They have also defined a test adequacy criterion for the dynamic slicing technique that is based on slice domain and boundary, slice coverage, full-predicate coverage and boundary testing. Dinh-Trong et al. [51] present a technique to generate test data to test design models using sequence diagrams. It uses a variable assignment graph (VAG) which is a directed graph that consists of nodes and edges such that the nodes record the change in values of variables and the edges represent the control flow during the execution of a sequence diagram.

Basanieri and Bertolino [52] propose the UIT (Use Interaction Testing) methodology to generate test cases by analyzing use case diagrams and interaction diagrams. Later, they proposed a test strategy, named CoWTeST (Cost Weighted TEST Strategy) [53], for selecting and prioritising test cases using the UIT methodology. They automated this strategy by implementing a tool, named CowSuite [54], that

generates test cases using use case and sequence diagrams. Pilskalns et al. [55] present an approach to generate test cases from sequence diagrams. They convert a sequence diagram into object method directed acyclic graph (OMDAG) such that its objects become the nodes and its method calls become the edges of the graph. The paths in the OMDAG are augmented with test information (different attribute values and parameter values of methods) that is used to generate test cases. The test cases generated using this approach validate the software model, and do not test the software implementation. Wittevrongel and Maurer [56] develop a model-based tool, SCENTOR, which creates functional test drivers for e-business applications from sequence diagrams that have test data (parameters and expected values of method calls) embedded in them. Fraikin and Leonhardt [57] develop another model-based tool, SeDiTeC, which generates test stubs using sequence diagrams that are augmented with test data. These stubs enable testing even before the completion of the system implementation. However, both these tools are model-based but not model-driven i.e., they do not take advantage of MDA technology. Lund et al. [58] present an algorithm to derive tests from models conforming to the UML 2.0 sequence diagram specification. A test is expressed as a sequence diagram consisting of only one lifeline. It operates by sending messages to the system under test and observing the messages received.

2.2 Descriptions of use case charts

Whittle [22] proposes an extension to the UML activity diagram, Use Case Chart (UCC), to support automation of software development activities, by specifying precise use cases and scenarios. A use case is defined as a particular use of a system, and groups the scenarios that have a common user goal. A scenario is sequence of steps describing interaction between user and system [31].

A UCC is a 3-level extension of the UML activity diagram. Level-1 is an activity diagram in which each node represents a use case. Level-2 is a set of activity diagrams such that there is an activity diagram for each use case in level-1. The nodes of a level-2 activity diagram represent different scenarios of a use case. Level-3 is a set of interaction diagrams such that there is an interaction diagram for each scenario in level-2.

Usually the requirements of a software system are written as use case diagrams and text-based templates [59]. This representation of requirements lacks formal semantics because the relationships among use cases or scenarios are not explicitly defined, which hinders automation of software development activities. The UCC complement use cases by adding these relationships [60]. The UCC can facilitate automated synthesis of state machines [59] and automated generation of test cases. The latter is one of the intended objectives of this project.

2.3 Model-driven testing

Model-driven testing uses the emerging MDA technology to automate testing activities. The MDA technology uses models, meta-models and transformation specifications to leverage automation of software development activities. In this approach, models are the basic software development artefacts. The meta-models are definitions that are used for interpreting models. The transformation rules are mappings between a source and a target meta-model. MDA uses model-transformation tools (MTT) to execute transformations defined in this way. By automating transformations, such as from activity diagrams to test cases, MDA-based tools can reduce development time and maintenance effort in model-based testing.

Dai discusses the transformation of a UML model into a UML 2.0 Testing Profile (U2TP) model [61]. U2TP is a general meta-model for testing, proposed by the Object Management Group (OMG). Dai proposes generating test cases using three transformations: i) UML model to a U2TP model, ii) the U2TP model to a platform-specific model (PSM) and iii) the PSM to a JUnit test case. However, no tool support is provided for the proposed method. Zander et al. [62] present a similar approach to transform a U2TP model into executable test cases for TTCN-3, which is a standardised test technology for test definition, implementation and execution [63]. They provide transformation rules between the source U2TP meta-model and the target TTCN-3 meta-model.

Dinh-Trong et al. [64] develop an Eclipse Plug-in for Testing UML Designs (EPTUD) that generates and executes test cases, using sequence diagrams. EPTUD transforms a UML model into an executable form (EDUT, executable design under test), adds test scaffolding (TDUT, testable design under test), executes tests and reports

failures. The test cases generated by EPTUD validate the UML model whereas the model-driven approach that we propose in this project verifies (tests) the implementation of CB software.

Engels et al. [65] present a model-driven monitoring approach in which assertions are used to monitor the behaviour (implemented by the developer) during execution. These assertions are generated from the contracts that are added to the model. These contracts represent the behaviour of the model and they consist of pre- and post-conditions of operations.

2.4 Component-based testing

Component-based (CB) testing is an important activity in CSD for developing reliable CBS. A major problem in CB testing is the lack of adequate information about the component, which makes it a challenging task (Section 1.4). Researchers deal with this problem by addressing the lack of information or the difficulty in testing [66]. Different techniques that are proposed for addressing these problems in CB testing are discussed below.

2.4.1 Built-in testing

Built-in testing (BIT) is a technique in which built-in tests are added in the component's code, to add support for testing the component [66]. BIT refers to all mechanisms that add information to a component's code for facilitating testing or checking assertions at runtime [14].

Wang et al. [67] propose a BIT approach for developing maintainable CB software in which built-in tests are added to the component's code such that the component user can decide whether to execute these tests or not. The component user can run the component in "test (maintenance) mode" or "normal mode". In test mode, the built-in tests are executed during execution of the component whereas in normal mode, these tests are not executed.

Wang's approach increases component size due to the added tests. To address this problem, Memotko and Zalewska [68] propose the Component+ BIT method which separates test cases from the component. The component provider produces a BIT-component and a test-component. The BIT-component is a component that has built-in

testing capabilities. The test-component contains test cases and interacts with the built-in testing capabilities of the BIT-component through its interfaces.

Beydeda and Gruhn [69] propose a self-testing strategy for COTS components (STECC). They suggest augmenting the test component with analysis functionality and testing tools. By doing this, the information that the component user needs to generate test cases can either be encapsulated in the component, or it can be generated on demand.

Edwards [70] suggests supporting the flow of information from the component developer to the component user using wrappers. In this approach, the component developer adds the information, that can help in CB testing, to the component and provides some wrappers that can interact with the component. The component user can use these wrappers to extract information from the component. The component user can add or remove wrappers from the component without having access to the source code.

BIT approaches increase testability of components by adding built-in tests to the component. However, they have the following drawbacks:

- i) they increase the size of the component, and
- ii) test cases are developer-oriented and the component user cannot influence the generation of test cases.

2.4.2 Component metadata

In the component metadata approach, the component developer equips the component with some information (component metadata) that the component user can use for performing CB testing. Component metadata can be either metadata or metamethods [71]. The metadata are information about the component and metamethods are the methods that retrieve or calculate information about the component.

Orso et al. [72] suggests all software engineering artefacts used for component development is metadata and should be shipped along with the component. The component provider can provide control-flow and data-flow graphs of the component that increase understandability and testability of the component. These graphs are also helpful in performing coverage analysis during CB testing.

Wu et al. [73] propose to deliver a UML model of the component as metadata. The UML model can be used to determine context-dependent relationships among the

components, which can be helpful for CB testing. Belli and Budnik [74] propose a similar approach in which they augment the component with UML statecharts. Test cases are generated from the UML statecharts, using model-based tools. Using these techniques, the component user can perform coverage-based execution of the model, to achieve greater reliability of the component. However, the component developer has to generate the model each time the component is modified.

Liu et al. [75] introduce the concept of retro-components. A retro-component has a retrospector in it which maintains testing and dynamic execution history. It records the tests that are conducted by the component developer and makes this testing information available to the component user. Retrospectors enhance the component such that the user can query the information provided and collect relevant information during their own testing activities.

In common with BIT, metadata approaches enhance the testability of components. They have the following advantages over BIT:

- i) test cases are not stored in the component's code and thus they do not increase the size of the component,
- ii) the component user can use the information, delivered as metadata, to generate dynamic test cases, and
- iii) metadata may support the generation of test data.

A disadvantage of this approach is the implementation transparency of the component, which is the hiding of implementation details from the component user.

2.4.3 Component certification

A component user is always concerned about the quality and reliability of a component. To increase the component user's trust, components can be certified before their reuse in CBS [76]. The following techniques have been proposed to certify components:

1. third-party certification,
2. developer certification, and
3. user certification.

Counsell [77] suggests that a component should be certified by a third-party. In third-party certification, an independent organisation tests the quality of the component and provides the test results, along with the test environment, to the component user.

Ma et al. [78] propose a framework for third-party certification that consists of following three steps:

- i) The third-party provides guidelines to the component developer.
- ii) The component provider generates a test package using these guidelines.
- iii) The third-party executes the test package and produces a test report.

An evaluation of this framework revealed some errors in a component, which demonstrates its usefulness. An advantage of third-party certification is that it is conducted by a neutral organisation, and hence the results are not biased.

The certification of a component through a third-party may be costly and small organisations may not be able to afford it. Morris et al. [23] propose that the component developer should perform component certification in order to avoid the cost associated with third-party certification. In this approach, the component developer attaches test cases along with their results (as a proof of their execution) to the component. An advantage of this approach is that the component user can determine, by examining the test cases, how thoroughly (adequately) the component developer has tested the component.

The test cases executed during developer certification are context-independent and test results may be biased as it is conducted by the component developer. To address this issue, Voas [79] proposes that the component user should certify the component using black-box testing, in which test cases are generated from the interface specifications of the component. The component user may use fault-injection techniques in which faults are generated instead of testing the component with the correct inputs, to determine the reliability of the component. Advantages of this approach are:

- i) The component user defines test requirements and thus the component is certified using context-dependent test oracles.
- ii) The test cases context-dependent

A disadvantage of this approach is that it does not address the third problem (Section 1.4) whereas the framework that we propose includes an approach for determining and extending the adequacy of testing at the time of component reuse.

2.4.4 Testable architecture

In the testable architecture approach, the component developer equips the component with an architecture that allows the component user to execute test cases.

Gao et al. [80] introduce testable beans to increase testability of a component. In this approach, the component developer implements an interface for testing (test interface) and codes test cases in the form of clients. The testable beans are components that are:

- i) Deployable and executable.
- ii) Traceable to allow the user to monitor and track their behaviour.
- iii) Testable by implementing test interfaces to access their self-test capabilities.
- iv) Usable with testing tools, i.e., they can interact with these tools.

An advantage of this approach is that the test cases are stored in clients and they are not part of the testable bean. However, the component developer has to do much work to maintain the testable beans.

Jabeen and Rehman [81] propose a framework for testing object-oriented components, in which the component developer, the component user and a third-party communicate test information using descriptors. These descriptors contain requirements of the component. The component developer prepares a component descriptor and attaches it to the component. The component user specifies component's requirement in another descriptor, the component requirement descriptor. The third-party generates test information using the information in the component descriptor and the component requirement descriptor. Advantages of this approach are:

- i) It allows everyone (the component developer, the component user and the third-party) to participate in CB testing.
- ii) It ensures that the component developer has provided the functionality that is required by the component user.

Testable architecture approaches do not increase the size of the component because the additional information is provided in the form of component's specifications and not within the component. However, similar to the metadata approach, they affect the implementation transparency of the component.

2.4.5 Discussion

A comparison of the approaches discussed in Section 2.4.4, is shown in Figure 1.

Approach	✓ Pros	✗ Cons
Built-in testing	1. Increase testability	1. Increase component size 2. Static test cases
Component metadata	1. Increase testability 2. Dynamic test cases 3. No increase in component size 4. Generate test data	1. Affect Implementation transparency
Testable architecture	1. Increase testability 2. No increase in component size	1. Affect Implementation transparency
Third-party certification	1. Impartial testing	1. May be too costly for small organisations to afford
Developer certification	1. Test cases are available to user to re-execute	1. Context-independent testing 2. Testing is biased by the component developer
User certification	1. Context-dependent testing	1. No test adequacy criteria for component reuse

Figure 2: Comparison of CB testing approaches

Test adequacy of the components, whose design and source code are not available to the component user, is an issue because:

- i) The component user cannot apply traditional test adequacy techniques such as coverage of models or source code.
- i) The component developer can use traditional CDCT techniques to certify the component. However, the test cases executed to certify the component are independent of the usage context of the component. Moreover, the testing can be biased.

We will use the component metadata that is used during developer certification to address the test adequacy of the component when it is reused in a new context.

3 Work Done: Generating unit test cases using model-driven architecture

In the work to date, we have proposed a model-driven method to generate test cases from UML diagrams [82]. As a case study we chose the generation of test cases for xUnit family members from sequence diagrams [31] that describe dynamic interactions among the components of a system.

xUnit [83] is a family of unit-testing frameworks used to write and run repeatable tests for software applications. Developers use these frameworks for developing and executing unit test cases, and for regression testing. Amongst the most popular family members of xUnit are JUnit [6] and SUnit [84], which are unit testing frameworks for testing Java and Smalltalk applications respectively.

The MDA-based method requires models, meta-models, transformation specifications and model transformation tools (MTT). Models are the basic artifacts that are manipulated for automating software development activities. The meta-models are the definitions of these models and they are used by MTT to interpret the models. Transformation specifications are the rules that specify the model transformations. The model transformation tools execute the transformation rules on the models to carry out their transformations. Model transformations can be horizontal transformations or vertical transformations. A horizontal transformation is one that maintains the abstraction level, e.g., a transformation from a Platform Independent Model (PIM) to another PIM. A vertical transformation is one that changes the abstraction level, e.g., a transformation from a PIM to a Platform Specific Model (PSM).

An overview of the approach is shown in Figure 3. The generation of test cases is performed in two steps. In the first step, a UML sequence diagram is translated into a testing model using a horizontal transformation. In the second step, the testing model is converted into a concrete and executable test case using a vertical transformation. We have two versions of the implementation (for JUnit and SUnit). The JUnit implementation is discussed in this section.

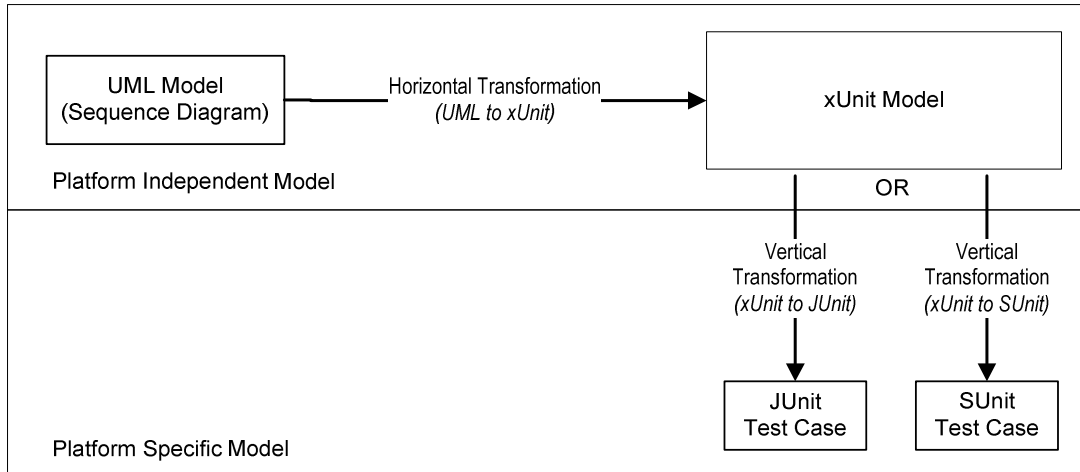


Figure 3: Overall Process

We test a system using sequence diagrams at two levels. The first is to generate test cases from a sequence of method calls that are selected by the tester from the sequence diagram. Typically, the selected method calls originate from a particular lifeline in the sequence diagram and they appear as method invocations in the generated test case. Note that method invocations that originate from subsequent lifelines are invoked indirectly by the selected method calls. To test that this happens as specified in the sequence diagram, we capture method execution traces during the execution of test cases to have a second check on the behaviour.

Test results are checked by comparing expected and actual return values of the selected method calls, and by comparing the execution traces with the method calls in the sequence diagram.

When this method was proposed and implemented, our objective was to generate unit test cases using sequence diagrams. Therefore we picked messages that originate from a particular lifeline to observe the behaviour originating from the lifeline. However, to use this method for CDCT, we can select the method calls that are directed to the component (the lifeline that represents the component in the sequence diagram), i.e., instead of selecting outgoing method calls, we focus on incoming calls to the component.

3.1 Generating Test Cases using MDA

The model-driven approach that we use for generating unit test cases consists of two steps. The first step is the creation of a test case which is generic to all xUnit family

members and the second step is the transformation of the generic test case into a concrete one, specific to a particular xUnit family member, e.g. a JUnit test case. In the first step, we model a sequence diagram as a sequence of methods calls (SMC) which is then automatically transformed into an xUnit model by applying model-to-model transformations using Tefkat [85]. In the second step, JUnit test cases are generated from the xUnit model by applying model-to-text transformations using MOFScript [86]. This process is shown in Figure 4.

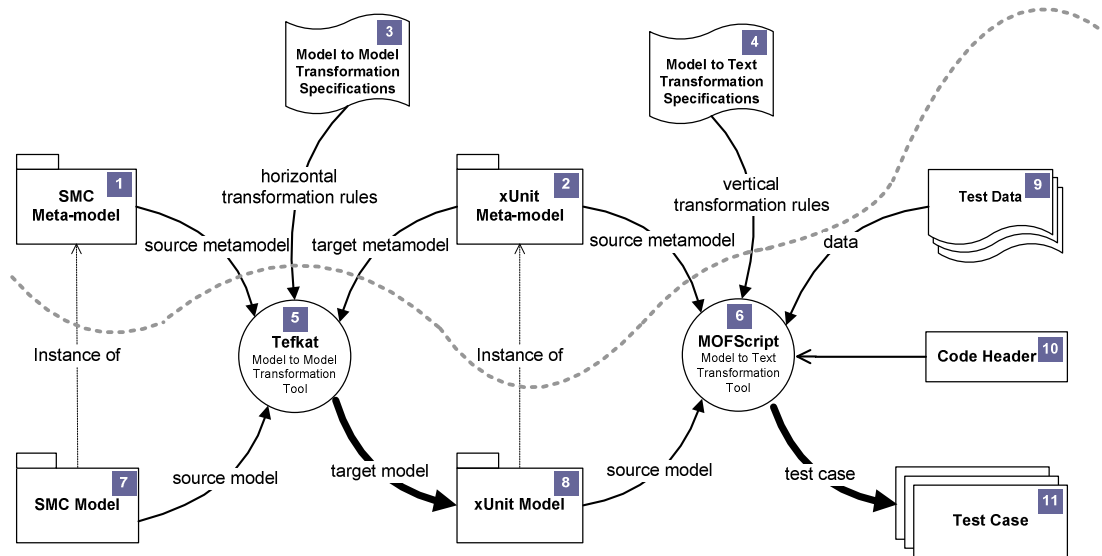


Figure 4: Overview of Methodology

Artifacts 1 and 2 in Figure 4 are the meta-models for a sequence of method calls and xUnit respectively, discussed in Sections 3.1.3 and 3.1.4. Artifacts 3 and 4 are transformation rules, discussed in Sections 3.1.1 and 3.1.2. Artifacts 5 and 6 are the two MDA tools that we use in our methodology to execute horizontal and vertical transformation specifications (Figure 3). Artifact 7 is the source model of the application from which we generate test cases. Artifact 8 is the xUnit model which is an intermediate output. Tefkat executes horizontal (model-to-model) transformations on the source model (7) to generate this xUnit model (8). Artifact 9 is the test data for the SMC model which specifies parameter values and expected return values of method calls in the sequence diagram. By changing the contents of the test data file, different test cases can be generated for the same sequence diagram. Artifact 10 is a simple text file containing code which is copied to the top of the file containing the test case. It can be used to define packages, import classes, etc. which are needed for compiling and

executing the generated test cases. Artifact 11 is the final output which is a concrete and executable unit test case produced by MOFScript. MOFScript reads the test data (9) and code header (10) while executing the vertical (model-to-text) transformations (4) on the xUnit model (8) to generate the unit test case (11).

These artifacts are generic at different levels. Artifacts 1, 2 and 3 are independent of platform, application and sequence diagrams. Artifact 4 is specific to a platform but independent of application and sequence diagram. None of these artifacts (1, 2, 3 and 4) need to be modified when testing different applications on the same platform. To test an application from a sequence diagram, the tester must provide the SMC model (7), test data (9) and the code header (10) file. Note that by altering the test data file, the same testing scenario can be executed with different test data.

We have implemented this methodology in a model-driven tool named MTCG (Model-driven Test Case Generator). MTCG is implemented using Tefkat and MOFScript and targets JUnit and SUnit.

3.1.1 Step 1: Transforming SMC into xUnit

We transform the SMC model into an xUnit model by using Tefkat transformation rules (Artefact 3 in Figure 4). As an example, the rule in Figure 5 creates a test case in the xUnit model for every SMC in the model. The test case is given the same name as the SMC. All the Tefkat rules that are used in the transformation are available online [87].

```

RULE SMC_2_TestCase ( smc, testCase )
  FORALL      SMC smc
  MAKE        TestCase testCase
  SET         testCase.name = smc.name ;

```

Figure 5: An Example Tefkat Rule

3.1.2 Step 2: Generating JUnit from xUnit

MOFScript transformation rules are used to generate JUnit test cases from the xUnit model. Two example MOFScript transformation rules are presented in Figure 6. The rule model.TestSuite::main is the entry-point rule where the transformation starts. The expression self.name is the name of the object on which the rule is being executed, i.e. the name of the test suite in this case. The forEach keyword iterates over the collection of test cases in the test suite and invokes the rule model.TestCase::mapTestCase to

process them. This rule creates a JUnit specific test case and invokes other rules (that are not discussed in detail) to complete the body of the test case.

```

model.TestSuite::main( ) {
    printf("public class Test_"+self.name+"extends TestCase {\n" )
    self.testCase->forEach ( tc:model.TestCase ) {
        tc.mapTestCase( )
    }
    printf( " } // End of Test Suite" )
}

model.TestCase::mapTestCase( ) {
    printf(" \n\r\t public void Test_" + self.name + "( ) { " )
    ...
    printf(" \n\r\t } //End of Test Case" )
}

```

Figure 6: Example MOFScript Rules

We have implemented MOFScript transformation rules for generating JUnit and SUnit test cases, from the xUnit model. All these rules are available online [87].

3.1.3 SMC Meta-model

As our focus was to investigate the use of models for automating software testing, we confined our implementation to a meta-model of sequences of method calls (SMC), ignoring more complex aspects of sequence diagrams [88] such as connectors, message-occurrence-specifications, message-ends and message-events.

Our meta-model for sequences of method calls is shown in Figure 7. It consists of interactions, messages, classes, parameters, expected values and literal strings. In this model, NamedElement represents a named value and LiteralString represents a string value. An Interaction represents part of a sequence diagram. The Messages contained in the interaction are a subset of the method calls of the sequence diagram selected by the tester. The messages can have Parameters and an optional ExpectedValue in them. The parameters and the expected value are of type ScalarValue or ComplexValue. The Scalar Values are atomic data values that do not contain any other data values. Instances of ScalarValues in Java are integer, float, String, etc. The ComplexValues are the values that contain other values, i.e., they act as data structures. The examples of ComplexValues in Java are all classes except Strings. Moreover every message is associated with an OwnerClass (that owns the methods being called), which is a class

that receives the message. The owner class has parameters for its constructors that are required to create an instance of the class in the generated test case.

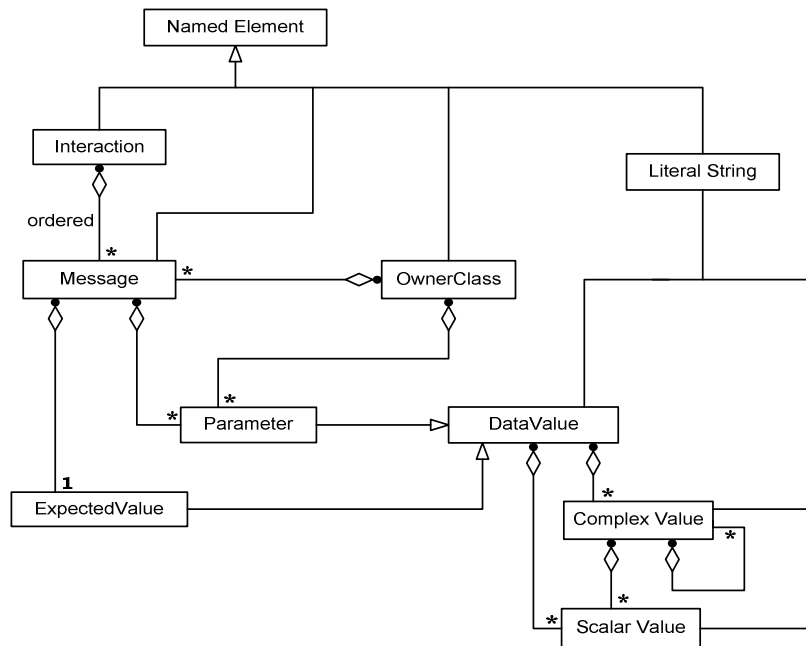


Figure 7: SMC Meta-model

3.1.4 xUnit Meta-model

The meta-model for xUnit test cases is shown in Figure 8. No meta-model for xUnit was available, so we derived it by studying the architecture of test cases written in different unit testing frameworks such as JUnit and SUnit. In this model, the Test Suite acts as a container for Test Case(s). A test case can have Assertions in it. An assertion is a condition that should hold true after executing the test case. An assertion can be of different types which are specified by its attribute type, e.g. the JUnit framework has Equal, Not Equal, Same, Not Same, True and False assertions. For testing of sequence diagrams, an assertion has a method call and an expected value. The method call is the code to be tested.

After executing a test case, the unit testing framework compares the actual value (the value returned after executing the code) with the expected value to decide on the success or failure of the test case. As an example, the JUnit's Equal assertion compares the actual value and the expected value. If both values are equal, the assertion holds. Conversely, the Not Equal assertion holds if the values are not equal. Moreover, the method can have parameters that are either scalar values or complex values as discussed

in the meta-model of SMC. The elements Message, OwnerClass, ComplexValue, ScalarValue and DataValue are the same as in the SMC meta-model.

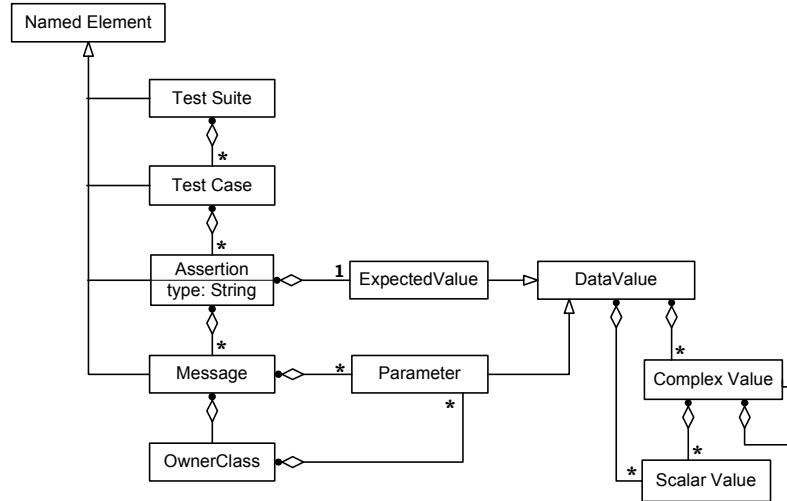


Figure 8: xUnit Meta-model

3.2 Tracing

During the execution of test cases, we monitor the method invocation chain by means of the Daikon [89] tracing tool. We compare the observed method execution chain with the expected method execution chain in the sequence diagram. Currently, we compare the traces manually, but this activity can be automated in the future.

3.3 An Example of Test Case Generation

MTCG has been implemented under Eclipse 3.1 [90]. It has been validated on an Automatic Teller Machine (ATM) simulation system [91]. An ATM allows its users to perform basic banking operations like withdrawal, deposit, transfer and checking their balance, without having to go to a bank. In an ATM, the user inserts an ATM card, enters a PIN (a personal identification number), selects a transaction to be performed and provides input needed for the transaction, e.g. amount and account in case of withdrawal. In response to the user's actions, the ATM reads the card, reads and validates the PIN, processes the transaction, dispenses cash, prints the receipt and ejects the card at the end of the session.

To test the ATM system, test cases were generated (using MTCG) from the following sequence diagrams: withdrawal, deposit, transfer and balance inquiry. These

sequence diagrams are derived from the information available with the ATM system [91]. For instance, the withdrawal sequence diagram shown in Figure 9 was based on a high-level collaboration diagram.

3.3.1 An Example Sequence Diagram

Figure 9 shows the sequence diagram for the withdrawal operation. The details like validation of the PIN and interaction with the bank are omitted to simplify the example. The sequence diagram consists of the following classes. The class Session represents a particular session, i.e. the operations that the user performs between inserting a card and the card being ejected. The Simulation class is used to represent interaction between the ATM system and the devices attached to it, which are the card reader, cash dispenser, and customer console. The class ATM represents a particular ATM terminal. The class Transaction is a base class that instantiates all the transactions. The class Withdrawal represents a withdrawal operation, which is instantiated when the user opts to withdraw an amount. The CardReader class reads the ATM card and ejects it at the end of the session. The CustomerConsole gets input from the user like amount, PIN etc. The class CashDispenser dispenses cash as a result of a valid withdrawal request from the user.

In this sequence diagram, the first three messages are setup messages that are required to create ATM, Session and Simulation objects. The messages to the devices, attached to the ATM system, are invoked on a Simulation object. The Simulation object delegates these messages to the representation classes of these devices. The message readCard() is invoked on CardReader objects and returns an ATM card object. The card object has an integer attribute. The message readPIN() reads the PIN from the CustomerConsole. It takes a string parameter promptMessage (that is displayed on the customer console) as a parameter and returns the pin (entered by the user) as an integer. The method makeTransaction() creates a Withdrawal object when the customer selects an amount. It takes ATM, session, card and PIN as parameters. The method performTransaction() reads the amount to withdraw and the account to withdraw from. It ensures that the amount is within the daily withdrawal limit and that the cash dispenser has enough cash to satisfy the request. It then dispenses the cash.

To generate the test case, we select the messages that originate from the high-level Session object (that are represented with bold arrows in Figure 9). They are

3.3.2 SMC Model

We create the input instance (SMC model) in the Eclipse editor [90], for the above SMC (in Figure 9), which is shown in Figure 10. The message SETUP-1 is a setup message that creates an ATM object with id, name, place and address as its constructor's parameters. Similarly the messages SETUP-2 and SETUP-3 create Simulation and Session objects. Setup messages do not generate any method calls in the test case. They are a part of configuration and environment setup. The rest of the messages have parameters and return values.

To keep the creation of the test data file simple, we define the sub-structure (attributes) of objects only once. Therefore, when an object that is not a simple data type is used in a message as an owner class or a parameter for the first time, its sub-structure is specified along with the types of its elements e.g. int, float or objects. The reason for this is that MTCG generates code for the object the first time it encounters the object and stores it in a hash table. Later when this object is referred to in a message call, its sub-structure does not need to be specified. For example, in Figure 10, when the object ATM is used for the first time in the message SETUP-1(), its id, place, name and address are specified as the constructor's parameters. But when this object is passed as a parameter in the method makeTransaction(), its sub-structure is not specified again. Whenever a reference is made to this object during method invocation, the variable stored in the hash table is retrieved. However, for an expected value, the sub-structure is specified each time, as a new object needs to be created with different data values in it.

3.3.3 xUnit Model

Tefkat applies the specified transformations to generate an output instance, in the form of an xUnit model, from the input instance. The output instance for the input instance in Figure 10 is shown in Figure 11. In the xUnit model, a test case is generated for each SMC. For each message in the sequence diagram, an assertion and a method (within the assertion) are generated. For each parameter in a message (in the source model), a parameter is created in the method (in the target model). Similarly for the return value and owner class (which is used to represent the class that the method belongs to),

corresponding elements in the target are created. The objects SETUP-1, SETUP-2 and SETUP-3 are created just for setting up the environment and we do not generate assertions for these objects in the generated test case.

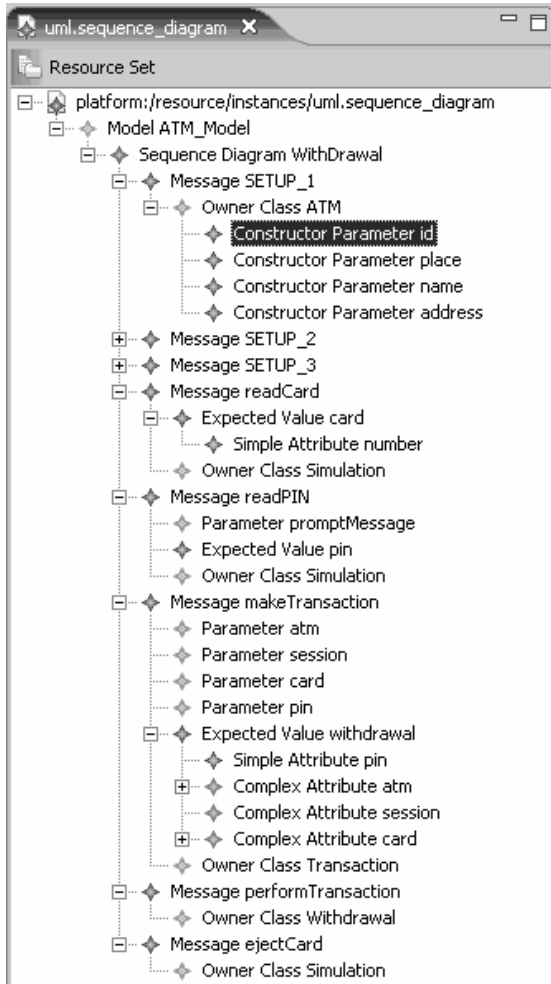


Figure 10: SMC Model

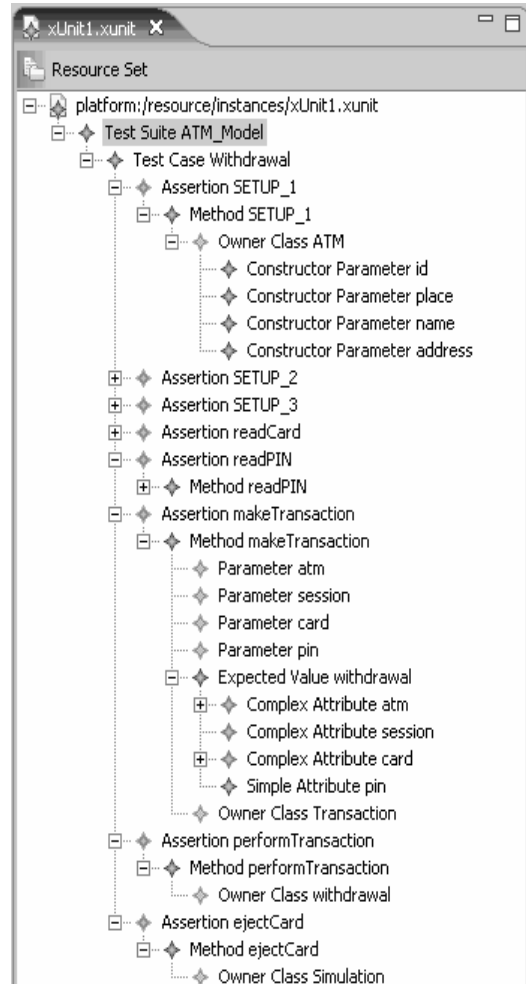


Figure 11: xUnit Model

3.3.4 Test Data File

The Test Data file that has the parameters and return values for the example in Figure 10 is shown in Figure 12. It has data for the calls to SETUP-1(), readCard() and readPIN() only. The reason is that the parameters for other method calls (e.g. the parameters atm, session, card and pin of the method makeTransaction) are created by previous methods.

The values for id, place, name, card number and pin (which are 41, Gorden College, National Bank, 1 and 42 respectively) in Figure 12 are provided by the tester.

```

<Sequence_Diagram name="Withdrawal">
  <Message name="SETUP_1">
    <Owner_Class name="atm">
      <Constructor_Parameter name="id" value="41"/>
      <Constructor_Parameter name="place" value="Gordon College"/>
      <Constructor_Parameter name="name" value="National Bank"/>
      <Constructor_Parameter name="address" value="null"/>
    </Owner_Class>
  </Message>
  <Message name="readCard">
    <Expected_Value name="Card" value="">
      <Simple_Attribute name="number" value="1">
    </Expected_Value>
    <Owner_Class name="Simulation"/>
  </Message>
  <Message name="readPIN">
    <Parameter name="promptMessage" value="Please enter PIN."/>
    <Expected_Value name="pin" value="42"/>
    <Owner_Class name="Simulation"/>
  </Message>
</Sequence_Diagram>

```

Figure 12: Test Data File

3.3.5 JUnit Test Case

The JUnit test case generated by MOFScript is shown in Figure 13. Lines 1-8 are copied by MOFScript from the Code Header file. The remaining lines are generated by MOFScript rules. Lines 12-16 create a JUnit test suite. Line 17 contains the test method for the sequence of calls identified. Lines 20-23 create variables that are used as the constructor's parameters for the ATM object atm. The data values for these variables, i.e. 41, "Gordon College" and "National Bank" and null, are read from the Test Data file (Figure 12). The types of these variables (i.e. int, String, InetAddress) are read from the xUnit model and originally come from the input model during the UML to xUnit transformation. Lines 20-25 are the code generated for the method SETUP-1(). As this is a setup message, it creates an ATM object for later use. The attribute values of the ATM object are read from the Test Data file. Lines 26-27 show the code generated for the methods SETUP-2() and SETUP-3() respectively. They generate Simulation and Session objects that use the previously created ATM object as their constructor's parameter. Lines 29-34 show the code generated for the call to readCard(). As this method returns a Card object, an expectedCard object is generated by reading the card number from the Test Data file.

```

1 package com.rnd;
2
3 import atm.*;
4 import atm.physical.*;
5 import atm.transaction.*;
6 import banking.*;
7 import simulation.*;
8 import junit.framework.*;
9
10 public class Test_Withdrawal extends TestCase {
11
12     public static void main( String args[] ) {
13         TestSuite testSuite = new TestSuite( Test_Withdrawal.class );
14         testSuite.run( new TestResult( ) );
15     }
16
17     public void test_Withdrawal( ) {
18
19         try {
20             int id = 41;
21             String name = "Gorden College";
22             String place = "National Bank";
23             InetAddress address = null;
24
25             ATM atm = new ATM ( id, name, place, address );
26             Simulation simulation = new Simulation ( atm );
27             Session session = new Session ( atm );
28
29             // Code for method readCard()
30             Card expectedCard = new Card();
31             int number = 1;
32             expectedCard.setNumber( number );
33             Card card = (Card) simulation.readCard();
34             assertTrue( expectedCard.equals( card ) );
35
36             // Code for method readPIN()
37             String parameterString1 = "Please enter PIN.";
38             int expectedPIN = 42;
39             int pin = simulation.readPIN( parameterString1 );
40             assertTrue( expectedPIN == pin );
41
42             // Code for method makeTransaction()
43             Withdrawal expectedWithdrawal = new Withdrawal (atm, session, card, pin);
44             Withdrawal withdrawal = (Withdrawal) Transaction.makeTransaction(atm,session,card, pin);
45             assertTrue( expectedWithdrawal.equals( withdrawal ) );
46
47             // Code for method performTransaction()
48             withdrawal.performTransaction();
49
50             // Code for method ejectCard()
51             simulation.ejectCard();
52
53         } catch ( Exception exp ) {
54             System.out.println( exp.toString() );
55             fail("Exception");
56         }
57
58     } //End of Method test_Withdrawal
59
60 } //End of Test Case Test_Withdrawal

```

Figure 13: JUnit Test Case

Lines 36-40 show the code generated for the call to `readPIN()`. This method returns `pin` as an integer value. A variable `expectedPIN` is generated whose value is read from the `Test Data` file. Lines 42-46 show the code generated for the method `makeTransaction()`. As it returns a `Withdrawal` object, an `expectedWithdrawal` object is created. Lines 47-48 show the code generated for the call to `performTransaction()` and lines 50-51 show the code generated for the call to `ejectCard()`. Assertions are generated in lines 34, 40 and 45 for the methods that return a value in order to compare them with the expected values.

The assertions in lines 34 and 45 use the `equals()` method, which is supplied by the tester, to compare the expected and the actual return value of a method. The `equals()` method is used for comparison of non-scalar values and user-defined types. For the comparison of scalar values, the “= =” operator is used for comparison, as shown in line 40.

3.3.6 Traces

While executing test cases, we capture their traces using the Daikon tracing tool [89]. The trace captured during execution of the test case in Figure 13 is shown in Figure 14. The method invocations with grey background are those that are not included in the test case but that are invoked by the methods that are called in the test case. This allows us to check that the classes interact as specified in the sequence diagram for this test case.

```
Simulation.readCard::ENTER
CardReader.readCard::ENTER
CardReader.readCard::EXIT
Simulation.readCard::EXIT

Simulation.readPIN::ENTER
CustomerConsole.readPIN::ENTER
CustomerConsole.readPIN::EXIT
Simulation.readPIN::EXIT

Transaction.makeTransaction::ENTER
Simulation.readMenuChoice::ENTER
CustomerConsole.readMenuChoice::ENTER
CustomerConsole.readMenuChoice::EXIT
Simulation.readMenuChoice::EXIT
Transaction.makeTransaction::EXIT

...
```

Figure 14: Execution Trace

3.4 Discussion

This method is model-driven, and uses model transformation technology. This is an advance over existing model-based testing approaches that do not take advantage of the emerging MDA technology. The overall process (Figure 3) is quite general in that it can be applied to different UML diagrams such use cases, sequence diagrams or state machines.

The genericity of our method is extended by targeting xUnit testing frameworks and incorporating an intermediate phase which generates test cases in a platform-independent xUnit format. Thus, potentially, the method can be used to generate test cases in any of the xUnit family by varying the backend (Artifact 4 in Figure 4). This demonstrates the versatility and utility of the MDA approach to software development and tool construction. It also distinguishes our tool from other tools, which typically generate test cases for one particular platform [56, 57].

Most of the vertical transformation rules (xUnit to SUnit and xUnit to JUnit) have similar logical structure that makes them reusable. They differ only in the text that is embedded in them, e.g. for SmallTalk the statement terminator is a dot (.) whereas in Java it is a semi-colon (;). The similarity of the logical structure of three example rules, mapAssertion, mapExpectedValue and mapMethod is illustrated in Figure 16. For the rules having similar structure, the implementer only needs to copy and change the language-specific syntax in these rules. Figure 15 shows the reusability in terms of non-commented lines of code that are the same. This reusability is obtained at minimal cost due to the intermediate xUnit model. The structural mapping between SMC and xUnit is addressed during the horizontal transformations, leaving the vertical transformations linear and almost identical except for language-specific syntax.

Rule	Structure	Lines of Code			
		xUnit-SUnit	xUnit-JUnit	Same	Reuse-%
main	Different	27	25	17	63
mapTestCase	Different	13	20	11	55
mapAssertion	Same	48	47	44	92
mapMessage	Same	32	28	25	78
mapOwnerClass	Same	8	8	8	100
mapParameter	Same	58	56	51	88
mapConstructorParameter	Same	54	52	48	89
mapExpectedValue	Same	36	34	30	83
mapComplexAttribute	Same	28	28	26	93
mapSimpleAttribute	Same	26	25	22	85
Total		330	323	282	85

Figure 15: Code Reusability Matrix

The SMC meta-model, the xUnit meta-model and the horizontal transformation are created only once and do not change for different platforms, systems and sequence diagrams. The developer needs to provide vertical transformations for each new platform. The tester needs to provide the SMC model, the test data file and the code header to generate test cases using this tool.

The MTCG is only a prototype. The limitations and the suggestions for its improvement are as follows:

- i) As our focus was to investigate the use of models for automating software testing, we devised our own meta-model for SMC instead of using the UML 2.0 meta-model of sequence diagrams that is very complex.
- ii) Each SMC model is created manually using the Eclipse editor. The creation of SMC models could be automated by reading sequence diagrams from their graphical representations.
- iii) The comparison of the actual trace with the sequence diagram is done manually but it can also be automated.

We shall use MTCG as a part of our model-driven framework for CDCT. In this framework, we shall devise test cases from the component usage model, and transform these test cases into concrete and executable test cases using MTCG.

For performing CDCT, tracing may not be possible for the components whose source code is not available. Moreover, to select SMCs for generating test cases using MTCG, we shall pick the method calls that end with the component interaction (component interface methods), in contrast to the example in Figure 9 where we selected the method calls that originated from a particular lifeline.

<p>xUnit-SUnit :: Assertion</p> <pre> 1. model.Assertion::mapAssertion() { 2. returnType = self.expectedValue.first().type.trim() 3. returnVariable = "return_" + returnType 4. self.method->forEach(m:model.Method m = self.method.first()) { 5. methodName = m.name.trim() 6. assertionType = m.owner.assertionType // assert 7. isSetup = m.name.startsWith("SETUP_") 8. className = m.ownerClass.first().name 9. classInstance = m.ownerClass.first().name.firstToLower() 10. m.mapMethod() 11. self.expectedValue->forEach(e:model.ExpectedValue) { 12. e.mapExpectedValue() 13. } 14. if(isStatic) classInstance = className 15. if(not isSetup) { // No method call for setup messages 16. text = "\n\t\t" 17. text = returnVariable + " := " + returnType + " new." 18. text = returnVariable + " := " + className + methodName + 19. text = text + "\n\t\t" + assertionType + ": (" 20. + expectedVariable + " equals " + returnVariable + ")." 21. } 22. outputFile.println(text) 23. } </pre>	<p>xUnit-JUnit :: Assertion</p> <pre> 1. model.Assertion::mapAssertion() { 2. returnType = self.expectedValue.first().type.trim() 3. returnVariable = "return_" + returnType 4. self.method->forEach(m:model.Method m = self.method.first()) { 5. methodName = m.name.trim() 6. assertionType = m.owner.assertionType // assertTrue 7. isSetup = m.name.startsWith("SETUP_") 8. className = m.ownerClass.first().name 9. classInstance = m.ownerClass.first().name.firstToLower() 10. m.mapMethod() 11. self.expectedValue->forEach(e:model.ExpectedValue) { 12. e.mapExpectedValue() 13. } 14. If(isStatic) classInstance = className 15. If(not isSetup) { // No method call for setup messages 16. text = "\n\t\t" 17. text = text + returnType + " " + returnVariable + " = (" + 18. returnType + ") " + className + " " + methodName 19. text = text + "\n\t\t" + assertionType + "(" + 20. expectedVariable + ".equals(" + returnVariable + "));" 21. } 22. outputFile.println(text) </pre>
<p>xUnit-SUnit :: Method</p> <pre> 1) model.Method::mapMethod() { 2) key = self.ownerClass.first().name 3) if(not key.equals("")) storedClassInstance = variableMap.get(key) 4) if(storedClassInstance.equals("")) { 5) if(isStatic == false) { // No method call for setup methods 6) text = "\n\t\t" 7) text = text + classInstance + " := " + className + " new." 8) } 9) outputFile.println(text) 10) variableMap.put(classInstance.trim(), classInstance.trim()) 11) } else { 12) classInstance = storedClassInstance 13) } 14) } </pre>	<p>xUnit-JUnit :: Method</p> <pre> 1. model.Method::mapMethod() { 2. key = self.ownerClass.first().name 3. if(not key.equals("")) storedClassInstance = variableMap.get(key) 4. if(storedClassInstance.equals("")) { 5. if(isStatic == false) { // No method call for setup methods 6. text = "\n\t\t" 7. text=text + classInstance+ " := " + className+" = new "+ 8. className+"();" 9. } 10. outputFile.println(text) 11. variableMap.put(classInstance.trim(), classInstance.trim()) 12. } else { 13. classInstance = storedClassInstance 14. } </pre>
<p>xUnit-SUnit :: Expected Value</p> <pre> 1. model.ExpectedValue::mapExpectedValue(scope) { 2. attributeType = self.type 3. attributeName = self.name 4. text = "\n\t\t" 5. text = text + attributeVariable + " := " + attributeType + " new." 6. text = "\n\t\t" + text + attributeVariable + " := " 7. text = text + "\n" 8. outputFile.println(text) 9. // Following line reads test data and appends it to the test case. 10. java ("com.m2t.WriteData", "writeTestData", scope, "C:/xunit2text") 11. text = "\n" 12. text = "." 13. outputFile.println(text) 14. } </pre>	<p>xUnit-JUnit :: Expected Value</p> <pre> 1 model.ExpectedValue::mapExpectedValue(scope) { 2 attributeType = self.type 3 attributeName = self.name 4 text = "\n\t\t" 5 text = text + attributeType + " " + attributeVariable + " = " 6 text = text + "\n" 7 outputFile.println(text) 8 // Following line reads test data and appends it to the test case. 9 java ("com.m2t.WriteData", "writeTestData", scope, "C:/xunit2text") 10 text = "\n" 11 text = "." 12 outputFile.println(text) 13 } </pre>

Figure 16: Rules for mapMethod, mapParameter and mapExpectedValue

4 Research Plan

This section discusses the proposed framework, the contributions of the project, the remaining tasks to be performed and gives timelines for those tasks.

4.1 The model-driven framework for context-dependent testing of components

We propose a framework to test a component for a new context, and to evaluate and extend the adequacy of the testing. The proposed framework consists of the following steps.

- i) Modelling component usage scenarios and generating test cases for CDCT
- i) Evaluating and extending test adequacy of CDCT
- ii) Executing the tests to test the component for the new context

i) Modelling component usage scenarios and generating test cases for CDCT

We shall model the component usage in the CB software using a use case chart (UCC) which consists of use cases, scenarios and interaction diagrams [22].

UCCs specify relationships among use cases and scenarios, as discussed in Section 2.2. Use cases and scenarios are often written in isolation without any explicitly defined relationship. UCCs add mechanisms to express relationships among them, to support automation of software development activities. Some advantages of using UCC for software testing are:

- i) UCCs contain interaction diagrams, that are associated with each scenario, which can be used to i) generate test cases for CB software, and ii) shift testing activities to an earlier part of CSD because the interfaces of components are available (or constructed) at an early stage of CSD.
- ii) Use cases and scenarios may have implicit dependencies on each other that can affect their behaviour. Therefore, use cases and scenarios should be run in different orders, during testing, to expose these dependencies. UCCs allow us to determine these execution orders of use cases and scenarios, because of their ability to define relationships between use cases and scenarios.

We shall generate test cases from the interaction diagrams, associated with each scenario of the UCC, for performing CDCT. We shall generate test cases only for those

use cases in which the component is used. Our test cases are a sequence of method calls (SMC) of a certain scenario (functionality).

ii) Evaluating and extending test adequacy of CDCT

We determine the adequacy of CDCT, using developer certification metadata (Section 2.4.3) [23]. We compare the test cases generated for CDCT with the test cases that were executed during developer certification, to determine the adequacy of CDCT, which is illustrated in Figure 17. *CDCT* denotes the test cases that are devised for performing CDCT, by the component user. *CT* denotes the test cases that were executed to certify the component, by the component developer. $CDCT \cap CT$ denotes the test cases that are common to *CDCT* and *CT*. $CT - CDCT$ denotes the test cases that are present in *CT* but not in *CDCT*. $CDCT - CT$ denotes the test cases that are present in *CDCT* but not in *CT*.

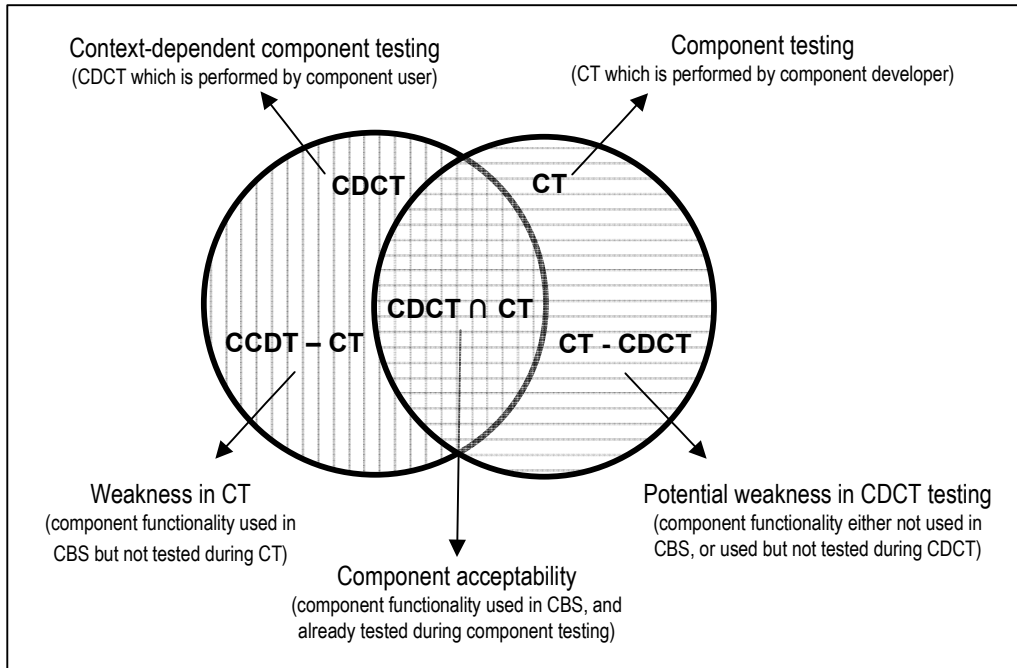


Figure 17: Adequacy of CDCT testing and component acceptability

The *CDCT* is considered to be adequate if it contains all the test cases that were executed during component testing, i.e., $CT - CDCT = \emptyset$. However, *CDCT* is potentially inadequate if some of the test cases in *CT* are not contained in *CDCT*, i.e., $CT - CDCT \neq \emptyset$. There can be two reasons for this:

- i) The test cases in $CT - CDCT$ are related to component functionality that is not used in the new context.

- ii) The test cases in $CT - CDCT$ are related to component functionality that is used in the new context, but they were overlooked by the component user while devising $CDCT$ for CB testing.

In situation (i), $CDCT$ is considered to be adequate. In situation (ii), $CDCT$ is considered to be inadequate.

The CT is considered to be adequate if all the $CDCT$ were already executed during component testing, i.e. $CDCT - CT = \emptyset$. However, there is a weakness in CT if some of the $CDCT$ were not executed during component testing, i.e., $CDCT - CT \neq \emptyset$. The adequacy of CT shows that the component is acceptable for the new context. Conversely, the component may not be acceptable if CT is inadequate. The discussion about the test adequacy of $CDCT$ and the component's acceptability, for the new context, is summarised in Figure 18.

Context-dependent component testing (CDCT)	
Adequate	$CT - CDCT = \emptyset$
Potentially inadequate	$CT - CDCT \neq \emptyset$
Component testing (CT)	
Adequate	$CDCT - CT = \emptyset$
Inadequate	$CDCT - CT \neq \emptyset$

Figure 18: Adequacy of CDCT and CT

If there is a potential inadequacy in $CDCT$ (i.e. the second case in Figure 18), we review the test cases in $CT - CDCT$. The purpose of this review is to identify those test cases that are present in $CT - CDCT$ and relate to component functionality that is used in the new context. We add those test cases to $CDCT$ to extend its test adequacy. This reduces the size of $CT - CDCT$.

If $CDCT - CT$ is non-empty, this raises a concern about the developer's component testing. This means that no CT was done for the relevant scenarios, so we need to be extra careful and should do extra testing in this area. As the test cases in $CDCT$ are generated from system-level use case scenarios, each of these test cases indicates an area of functionality of the component that was not tested in CT. In this case, we expand these test cases (that are generated for $CDCT$) to target the functionality of the component that was not tested before.

The implementation of the concepts presented in Figure 17 requires a comparison of test suites, and hence of test cases. We are not aware of established

methods for doing this. Therefore, we shall devise our own criteria to compare test cases and test suites. Then we shall evaluate these criteria and approaches to determine which is the most suitable for our purposes.

Initially, we propose to compare test cases, expressed as SMCs, based on the following criteria.

- i) Same Sequence of Method Calls (SMCs): Two test cases are considered the same if and only if they contain the same method calls in the same order.
- ii) Same Parameter Values: Two test cases are considered the same if and only if they contain the same method calls in the same order with the same parameter values for those methods.
- iii) Same Equivalence Class of Parameter Values: Two test cases are considered the same if and only if they contain the same method calls in the same order, and the parameter values for those methods belong to the same equivalence class [24]. These equivalence classes will be defined by the tester.

A simple comparison of two test suites, in which each test case of the first test suite is compared with each test case of second the test suite, may be too expensive and not scalable. We shall investigate other approaches that involve synthesis of state machines from test sequences [92] and comparing the state machines [93].

iii) Executing the test cases to test the component for the new context

We then transform these test cases, devised for CDCT, into concrete and executable test cases using MTCG. Finally, we execute these concrete test cases and examine the behaviour of the component for the new context. We do this by checking the values returned by interface methods (of the component) that are invoked in the test cases.

For a component whose design and source code are available, we can monitor the sequence of methods called during execution using the Daikon [89] tracing tool (as discussed in Section 3.2). We compare these sequences of method calls (SMC) against the SMC specified in the sequence diagram (of that scenario) to ensure the correct sequence of method invocations in the CB software.

4.2 Contributions

The contributions of this research will be:

- i) A novel model-driven framework for context-dependent testing of components
- ii) A novel algorithm for generating test case sequences from use case charts
- iii) A novel method for evaluating and extending test adequacy of CDCT
- iv) A novel criteria for comparing test cases and test suites
- v) Prototype tool support for the framework
- vi) Evaluation of the framework and tool support

i) A novel model-driven framework for context-dependent testing of components

The novelty of this framework is that it will apply the emerging MDA technology to CDCT. The proposed framework will benefit from the advantages of an MDA-based approach, such as portability, interoperability and maintainability. Part of this framework has already been implemented and was discussed in Section 3 [87].

ii) A novel algorithm for generating test sequences from use case charts

As UCCs are relatively new, they have only been used for model synthesis and system simulation purposes so far. They have not been explored in the context of automated software testing. We shall use the ability of UCCs to define relations among use cases and scenarios, to generate sequences of test cases within the test suites. These test sequences are the different orders that the scenarios can run in.

iii) A novel method for evaluating and extending test adequacy of CDCT

The adequacy of CDCT will be determined by comparing the tests generated from the UCCs with the tests executed during developer certification. We then analyse the test cases, executed by the component developer during component testing, to determine any weaknesses of CDCT and to extend its adequacy.

iv) Criteria for comparing test cases and test suites

Our method for evaluating and extending the adequacy of CDCT is based upon comparison of two test suites, and there are a number of ways to do this. We shall investigate these algorithms to determine the most appropriate one for our purposes.

v) Prototype tool support for the framework

We shall develop a prototype tool for the proposed framework. MTCG (a model-driven tool for generating test cases), which is a part of the prototype tool support, has been implemented and was discussed in Section 3. We implement the algorithms for comparing test cases and test suites.

vi) Evaluation of the framework and tool support

We shall evaluate our framework on two case studies. We shall use one case study to test the development of the framework and tool support. The other case study will be used to evaluate the completed framework and its tool support. For this purpose, we shall define evaluation criteria and collect data about the defect-detection rate and coverage achieved during the execution of the generated test cases. Finally, the data collected will be analysed and the framework will be refined if necessary.

4.3 Tasks

The tasks required to complete the thesis are grouped as follows.

- ◆ Development of the framework and its tool support
- ◆ Evaluation of the framework and its tool support
- ◆ Thesis Writing

Development of the framework and its tool support

1. Defining a case study for developing the framework and prototype tool support
2. Devising an algorithm for generating test sequences from use case charts
3. Proposing and evaluating criteria for comparing test cases and test suites
4. Providing prototype tool support for the proposed framework
5. Applying the framework (and its tool support) to the case study to test the development of the framework

Evaluation of the framework and its tool support

6. Defining a case study for evaluating the framework and the tool support
7. Evaluating the framework and the tool support on the case study - evaluation criteria will be defined, and the framework and tool support will be evaluated accordingly.

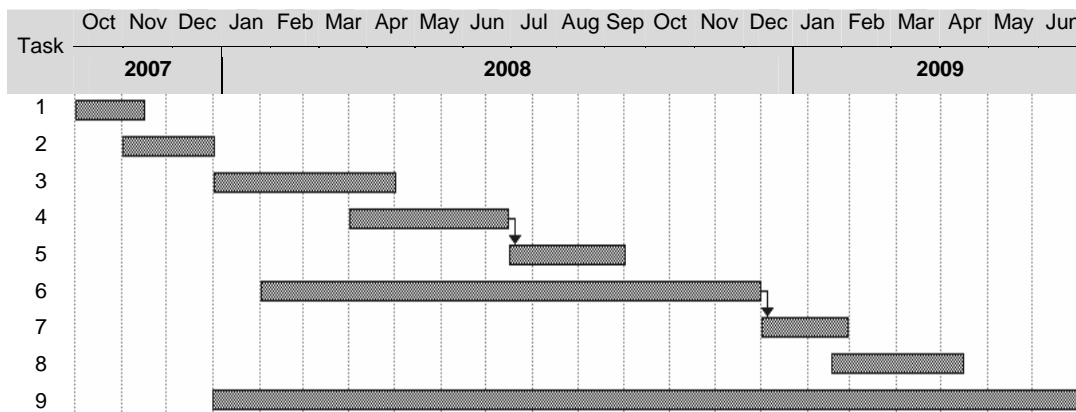
8. Analysis of the data collected during evaluation to determine the usefulness of the proposed framework and the tool support

Thesis Writing

9. Parts of the thesis will be written up as the work progresses

4.4 Timeline

The proposed timeline for the tasks listed in Section 4.3 is given below.



References

- [1] Object Management Group, "MDA Guide Version 1.0.1," 2004.
- [2] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture - Practice and Promise*. Addison-Wesley. 2003.
- [3] J. Bézivin and N. Ploquin, "Combining the Power of Meta-Programming and Meta-Modeling in the OMG/MDA Framework," Proceedings of the 2nd Workshop on UML for Enterprise Applications, 2001.
- [4] M. Rutherford and A. Wolf, "A case for test-code generation in model-driven systems," Proceedings of the 2nd International Conference on Generative Programming and Component Engineering, 2003, Springer-Verlag, pp. 377-396.
- [5] J. Poole, "Model-Driven Architecture: Vision, Standards and Emerging Technologies," Proceedings of the 15th European Conference on Object Oriented Programming, 2001, Springer-Verlag.
- [6] J. Rainsberger, *JUnit Recipes : practical methods for programmer testing*. Manning Publications Co. 2005.
- [7] I. Crnkovic, S. Larsson, and J. Stafford, "Component-based software engineering building systems from components," Proceedings of the 26th Annual International Computer Software and Applications Conference, 2002, IEEE Computer Society, pp. 816-817.
- [8] A. Andrews, S. Ghosh, and C. E. Man, "A model for understanding software components," Proceedings of the International Conference on Software Maintenance, 2002, IEEE Computer Society, pp. 359-368.
- [9] C. Szyperski, *Component software: beyond object-oriented programming*. Addison-Wesley. 1998.
- [10] J. Bosch, C. Szyperski, and W. Weck, "Summary of the Second International Workshop on Component-Oriented Programming," Proceedings of the 2nd International Workshop on Component-Oriented Programming, 1997.
- [11] S. C. Reid, "BS 7925-2: the software component testing standard," Proceedings of the 1st Asia-Pacific Conference on Quality Software, 2000, IEEE Computer Society, pp. 139-148.
- [12] O.-J. Dahl, E. Dijkstra, and C. Hoare, "Notes on Structured Programming," in *Structured Programming*. 1972. Academic Press, pp. 1–82.
- [13] L. Briand, Y. Labiche, and M. Sowka, "Automated, contract-based user testing of commercial-off-the-shelf components," Proceedings of the 28th International Conference on Software engineering, 2006, ACM Press, pp. 92-101.
- [14] H. Gross, *Component-Based Software Testing with UML*. Springer-Verlag. 2005.
- [15] E. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Softw.*, vol. 15, 1998, pp. 54-59.

- [16] M. Rehman and F. Jabeen, "Testing software components for integration: a survey of issues and techniques," *Softw. Test. Verif. Reliab.*, vol. 17, 2007, pp. 95-133.
- [17] J. Gao, H.-S. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*. 2003.
- [18] R. S. Freedman, "Testability of Software Components," *IEEE Transactions on Software Engineering*, vol. 17, 1991.
- [19] J. Gao and M. Shih, "A component testability model for verification and measurement," Proceedings of the 29th Annual International Computer Software and Applications Conference, 2005, IEEE Computer Society, pp. 211-218.
- [20] D. S. Rosenblum, "Adequate Testing of Component-Based Software," Dept. of Computer Science, University of California, Technical Report TR 97-34, 1997.
- [21] M. Delamaro, J. Maidonado, and A. Mathur, "Interface Mutation: an approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, 2001, pp. 228-247.
- [22] J. Whittle, "Specifying precise use cases with use case charts," in *Lecture Notes in Computer Science*, vol. 3844, *Satellite Events at MODELS 2005 Conference*. 2005. Springer-Verlag, pp. 290-301.
- [23] J. Morris, G. Lee, K. Parker, G. Bundell, and C. Lam, "Software Component Certification," *Computer*, vol. 34, 2001, pp. 30-36.
- [24] G. Myers, *The Art of Software Testing*. Wiley Interscience. 1979.
- [25] R. Dolores and U. Roger, "Software Verification and Validation: An Overview," *IEEE Softw.*, vol. 6, 1989, pp. 10-17.
- [26] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz, "Model-based testing in practice," Proceedings of the 21st International Conference on Software Engineering, 1999, IEEE Computer Society, pp. 285-294.
- [27] L. Apfelbaum and J. Doyle, "Model Based Testing," Software Quality Week Conference, 1997, pp. 296-300
- [28] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Elsevier Inc. 2007.
- [29] C. E. Williams, "Software Testing and the UML," International Symposium on Software Reliability Engineering, 1999.
- [30] R. Heckel and M. Lohmann, "Towards Model-Driven Testing," in *Electronic Notes in Theoretical Computer Science*, vol. 82, 2003.
- [31] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley. 1999.
- [32] M. Badri, L. Badri, and M. Naha, "A Use Case Driven Testing Process: Towards a Formal Approach Based on UML Collaboration Diagrams," in *Lecture Notes in Computer Science*, vol. 2931. 2004. Springer Berlin, pp. 223-235.

- [33] C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, 2006, pp. 140-155.
- [34] R. Eshuis and R. Wieringa, "Tool support for verifying UML activity diagrams," *IEEE Transactions on Software Engineering*, vol. 30, 2004, pp. 437-447.
- [35] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from UML activity diagram based on Gray-box method," Proceedings of 11th Asia-Pacific Software Engineering Conference, 2004, IEEE Computer Society, pp. 284-291.
- [36] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for UML activity diagrams," Proceedings of the 1st International Workshop on Automation of Software Testing, 2006, ACM Press, pp. 2-8.
- [37] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI testing using a model-driven approach," Proceedings of the 1st International Workshop on Automation of Software Test, 2006, ACM Press, pp. 9-14.
- [38] W. Shen, K. Compton, and J. Huggins, "A Toolset for Supporting UML Static and Dynamic Model Checking," Proceedings of the 26th International Computer Software and Applications Conference, 2002, IEEE Computer Society, pp. 147-152.
- [39] J. Lilius and I. Paltor, "vUML: a tool for verifying UML models," Proceedings of the 14th International Conference on Automated Software Engineering, 1999, IEEE Computer Society, pp. 255-258.
- [40] A. Tsiolakis, "Integrating Model Information in UML Sequence Diagrams," in *Electronic Notes in Theoretical Computer Science*, vol. 50. 2001. Elsevier, pp. 1-9.
- [41] L. Bousquet, H. Martin, and J.-M. Jezequel, "Conformance Testing from UML Specifications Experience Report," Proceedings of the Workshop of the pUML-Group held together with the Workshop on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists, 2001, GI Press, pp. 43-55.
- [42] A. Cavarra, C. Crichton, and J. Davies, "A method for the automatic generation of test suites from object models," *Information and Software Technology*, vol. 46, 2004, pp. 309-14.
- [43] S. Kansomkeat and W. Rivepiboon, "Automated-generating test case using UML statechart diagrams," Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, 2003 pp. 296-300.
- [44] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha, "Test cases generation from UML state diagrams," *Software, IEE Proceedings*, vol. 146, 1999, pp. 187-192.
- [45] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," Proceedings of the 2nd International Conference on Unified Modelling Language, 1999, IEEE Computer Society.

- [46] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation," Proceedings of the 3rd International Conference on Unified Modeling Language, 2000, Springer-Verlag.
- [47] S. Supavita and T. Suwannasart, "Testing polymorphic interactions in UML sequence diagrams," Proceedings of the International Conference on Information Technology: Coding and Computing, 2005, IEEE Computer Society, pp. 449-454.
- [48] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models," *Software Testing, Verification and Reliability*, vol. 13, 2003, pp. 95-127.
- [49] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Softw Syst Model*, vol. 1, 2002, pp. 10-42.
- [50] P. Samuel, R. Mall, and S. Sahoo, "UML Sequence Diagram Based Testing Using Slicing," IEEE Indicon Conference, 2005, pp. 176-178.
- [51] T. Dinh-Trong, S. Ghosh, R. France, and A. Andrews, "Generating test data to test UML Design Models," Proceedings of the 2nd MoDeVa workshop - Model design and Validation, 2005.
- [52] F. Basanieri and A. Bertolino, "A Practical approach to UML-based derivation of integration tests," Proceedings of the Fourth International Software Quality Week Europe and International Internet Quality Week Europe, 2000.
- [53] F. Basanieri, A. Bertolino, and E. Marchetti, "CoWTeSt: A Cost Weighed Test Strategy," Proceedings of the ESCOM-SCOPE, 2001, pp. 387-396.
- [54] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects," Proceedings of the 5th International Conference on the Unified Modeling: the Language and its Applications, 2002, Springer-Verlag, pp. 383-397.
- [55] O. Pilskalns, A. Andrews, R. France, and S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations," in *Lecture Notes in Computer Science*, vol. 2863. 2003. Springer, pp. 234-248.
- [56] J. Wittevrongel and F. Maurer, "SCENTOR: scenario-based testing of e-business applications," Proceedings of the 10th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001, IEEE Computer Society, pp. 41-46.
- [57] F. Fraikin and T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," Proceedings of the 17th IEEE International Conference on Automated Software Engineering, 2002, IEEE Computer Society, pp. 261-266.
- [58] M. Lund and K. Stolen, "Deriving tests from UML 2.0 sequence diagrams with neg and assert," Proceedings of the 2006 International Workshop on Automation of Software Test, 2006, ACM Press, pp. 22-28.
- [59] J. Whittle and P. Jayaraman, "Generating hierarchical state machines from use case charts," Proceedings of the 14th IEEE International Requirements Engineering Conference, 2006, IEEE Computer Society, pp. 16-25.

- [60] J. Whittle, "A formal semantics of use case charts," *Satellite Events at MODELS 2005 Conference, Lecture Notes in Computer Science*, vol. 3844, 2005, pp. 290-301.
- [61] Z. Dai, "Model-Driven Testing with UML 2.0," Proceedings of the 2nd European Workshop on Model Driven Architecture, 2004.
- [62] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din, "From U2TP models to executable tests with TTCN-3: an approach to model driven testing," in *Lecture Notes in Computer Science*, vol. 3502. 2005. Springer-Verlag, pp. 146-158.
- [63] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, W. Anthony, and W. Colin, "An introduction to the testing and test control notation (TTCN-3)," *Comput. Networks*, vol. 42, 2003, pp. 375-403.
- [64] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews, "A tool-supported approach to testing UML design models," Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005, IEEE Computer Society, pp. 519-528.
- [65] G. Engels, B. Güldali, and M. Lohmann, "Towards Model-Driven Unit Testing," Proceedings of the 3rd Workshop on Model Design and Validation, 2006, pp. 16 - 29.
- [66] S. Beydeda and V. Gruhn, "State of the art in testing components," Proceedings of the 3rd International Conference on Quality Software, 2003, IEEE Computer Society, pp. 146-153.
- [67] W. Yingxu, G. King, and H. Wickburg, "A method for built-in tests in component-based software maintenance," Proceedings of the 3rd European Conference on Software Maintenance and Reengineering, 1999, IEEE Computer Society, pp. 186-189.
- [68] J. Hornstein and H. Elder, "Test reuse in CBSE using built-in tests," Proceedings of the Workshop on Component-based Software Engineering, Composing systems from components, 2002.
- [69] S. Beydeda and V. Gruhn, "Merging components and testing tools: the self-testing COTS components (STECC) strategy," Proceedings of the 29th Euromicro Conference, 2003, IEEE Computer Society, pp. 107-114.
- [70] S. H. Edwards, "Toward reflective metadata wrappers for formally specified software components," Proceedings of the Workshop on Specification and Verification of Component Based Systems, held in conjunction with OOPSLA 2001, 2001.
- [71] A. Orso, H. Do, G. Rothermel, M. Harrold, and D. Rosenblum, "Using component metadata to regression test component-based software," *Softw. Test. Verif. Reliab.*, vol. 17, 2007, pp. 61-94.
- [72] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, "Using component metacontent to support the regression testing of component-based software," Proceedings of the International Conference on Software Maintenance, 2001, IEEE Computer Society, pp. 716-725.

- [73] Y. Wu, M. H. Chen, and J. Offutt, "UML-based integration testing for component-based software," 2003, Springer, pp. 251-60.
- [74] F. Belli and C. Budnik, "Towards Self-Testing of Component-Based Software," Proceedings of the 29th Annual International Computer Software and Applications Conference, 2005, IEEE Computer Society, pp. 205-210.
- [75] C. Liu and D. Richardson, "Software components with retrospectors," International Workshop on the Role of Software in Testing and Analysis, 1998, pp. 63-68.
- [76] A. Alvaro, E. Almeida, and S. Meira, "Software Component Certification: A Survey," Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005, IEEE Computer Society, pp. 106-113.
- [77] W. Councill, "Third-Party Testing and the Quality of Software Components," *IEEE Softw.*, vol. 16, 1999, pp. 55-57.
- [78] Y. Ma, S. Oh, D. Bae, and Y. Kwon, "Framework for Third Party Testing of Component Software," Proceedings of the 8th Asia-Pacific on Software Engineering Conference, 2001, IEEE Computer Society.
- [79] J. Voas, "Certifying off-the-shelf software components," *Computer*, vol. 31, 1998, pp. 53-59.
- [80] J. Gao, K. Gupta, S. Gupta, and S. Shim, "On Building Testable Software Components," Proceedings of the 1st International Conference on COTS-Based Software Systems, 2002, Springer-Verlag, pp. 108-121.
- [81] F. Jabeen and M. Rehman, "A framework for object oriented component testing," Proceedings of the IEEE Symposium on Emerging Technologies, 2005, IEEE Computer Society, pp. 451-460.
- [82] A. Javed, P. Strooper, and G. Watson, "Automated Generation of Test Cases Using Model-Driven Architecture," Proceedings of the 2nd International Workshop on Automation of Software Test, held at the 29th International Conference on Software Engineering, 2007, IEEE Computer Society, pp. 3.
- [83] P. Hamill, *Unit Test Frameworks*. O'Reilly Media. 2004.
- [84] SUnit: <http://sunit.sourceforge.net/>, accessed on 07/08/07.
- [85] Tefkat: <http://tefkat.sourceforge.net/>, accessed on 07/08/2007.
- [86] MOFScript: <http://www.eclipse.org/gmt/mofscript/>, accessed on 07/08/2007.
- [87] MTCG: <http://www.itee.uq.edu.au/~abuzafer/>, accessed on 07/08/2007.
- [88] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *Computer*, vol. 39, 2006, pp. 59-66.
- [89] Daikon: <http://pag.csail.mit.edu/daikon/>, accessed on 07/08/2007.
- [90] EMF: <http://www.eclipse.org/emf/>, accessed on 07/08/2007.

- [91] ATM System: <http://www.math-cs.gordon.edu/courses/cs211/ATMExample/>, accessed on 07/08/2007.
- [92] T. Maier, "Generation of complex statecharts from sequence diagrams," Master Thesis, Braunschweig University, 2002.
- [93] L. Michel and C. Eduard, "Comparing Generic State Machines," Proceedings of the 3rd International Workshop on Computer Aided Verification, 1991, Springer-Verlag.