ATAC

State-of-the-art document:

Processes and Methods for Advanced Test Automation of Complex Software-Intensive Systems

Project number:	ITEA 2 10037
Edited by:	Pekka Aho
Date:	26 August 2014
Document version no.:	4.0

This document contains an overview of the State-Of-The-Art as is known to the partners in the ATAC consortium. It has been continuously updated during the course of the ATAC project with a) new external evolutions in the field of test automation and b) the improvements resulting from the R&D&I activities executed by the ATAC project partners. Only a concise treatment of the latter in given in this document. More in depth details on the obtained results can be found in the different publications listed on the ATAC project website: http://www.atac.testautomation.fi/index.php?page=publications

HISTORY

Document version #	Date (yyyy/mm/dd)	Remarks
1.0	2012/10/11	The first release version of the deliverable
2.0	2013/10/01	The second release version of the deliverable
3.0	2014/05/28	The third release version of the deliverable
4.0	2014/08/21	The fourth release version of the deliverable

AUTHORS

Name:	Affiliation:	E-mail:
Pekka Aho	VTT	pekka.aho@vtt.fi
Teemu Kanstrén	VTT	teemu.kanstren@vtt.fi
Kivanc Doganay	SICS	kivanc@sics.se
Eduard Paul Enoiu	MDU	eduard.paul.enoiu@mdh.se
Antti Jääskeläinen	TUT	antti.m.jaaskelainen@tut.fi
Šarunas Packevičius	KTU	Sarunas@ieee.org
Virginija Limanauskienė	KTU	virginija.limanauskiene@ktu.lt
Sigrid Eldh	Ericsson	sigrid.eldh@ericsson.com

TABLE OF CONTENTS

1. Introduction

1.1. About this document

This document is a work in progress that aims to present to project partners and the European industry the state of the art in test automation in the scope of the ITEA2/ATAC (Advanced Test Automation for Complex Software-Intensive Systems) project. The document will be updated periodically as our research advances and as we analyse new issues related to test automation. The authors welcome requests for new topics to be assessed in the document.

1.2. Motivation

In a typical commercial development organization, the cost of providing assurance that the program will perform satisfactorily in terms of its functional and non-functional specifications within the expected deployment environments via appropriate debugging, testing, and verification activities can easily range from 50 to 75 per cent of the total development cost [Halpern2002]. Many of these software testing activities can be automated and with the help of test automation reduce the time to market and increase the quality of the resulting systems.

Although computing has got faster, smarter and cheaper, it has also become much more complex. Complexity seems to be unavoidably associated with software. Programming is said to be all about suffering from ever-increasing complexity [Hinchey2012]. There are various reasons for software complexity: software can be distributed, configurable, or it can be simply large. The two main activities against complexity are abstraction and decomposition [Hinchey2012]. Abstraction means hiding the complexity that is not relevant for the specific system or problem, and decomposition means breaking down the system or problem into parts that are easier to understand. Both abstraction and decomposition can be used also for enabling test automation for the complex systems. For example model-based testing (MBT) utilizes abstraction, and decomposition is used at many levels of testing.

Test Automation should be thought of regarding all aspects of the software testing, verification and validation during the whole life-cycle of the software. It is not sufficient to view and automate only the test execution, even though it is often the first approach in industry to make testing more efficient for large complex systems. Equally important is to automate other aspects, such as test case generation, visualization of results, etc.

2. Model-Based Testing (MBT)

Model-based testing (MBT) is a technique of generating test cases from models describing some relevant aspect of the system under test (SUT) at a chosen abstraction level. The idea is to provide more cost-effective means for extensive testing of complex systems. Instead of manually writing a large set of test cases, a smaller set of test models are built to describe generally the behaviour of the SUT and how it should be tested. A test generator tool is then used to automatically generate test cases from these models. There are several benefits, including easier test maintenance due to fewer artefacts to update, higher test coverage from the generated test cases, and documenting the SUT behaviour in higher level models which helps in sharing the information and understanding the system [Utting2006].

MBT process can be divided into five main steps: modelling the SUT and/or its environment, generating abstract test cases from the models, concretizing the abstract tests into executable tests, executing the tests on the SUT and assigning verdicts, and analysing the test results [Utting2006]. MBT is heavily dependent on tools. A wide variety of MBT tools exist, each with their own set of features and test generation algorithms [Utting2012]. Some may be better suited in their supported modelling approaches than others with respect to a chosen domain. Usually some kind of MBT tool is used to generate abstract test cases from a behavioural model of the SUT. Many of the MBT tools allow test engineer to control the focus and number of the test cases, and transforming the abstract tests into executable test scripts often requires some input from the test engineer [Utting2006].

MBT methodologies can be divided into two categories based on how the generated tests are executed: off-line testing and online testing [Utting2006]. In off-line testing the tests are first generated in their entirety, and the resulting test cases are then executed in a separate step. This approach fits well into traditional testing processes, with model-based testing simply replacing manual test creation. In contrast, online testing executes the tests as they are being generated. The advantage is that test generation can react to unexpected events in execution, making testing of non-deterministic systems easier. However, the lack of separate test cases necessitates greater changes to the entire testing process. Perhaps due to these difficulties, most work on MBT has focused on off-line testing. Examples of MBT tools capable of online testing include TorX [Tretmans2002] and TEMA [Jääskeläinen2009].

While MBT has been a topic of research for a long time, and in the recent years has started to see also more industrial adoption. Case studies in different domains include aerospace [Blackburn2002], automotive [Bringman2008], [Pretschner2005], medical [Vieira2008], communication protocols [Grieskamp2010], file systems [Miller2012], and information systems [Santos-Neto2008]. When dealing with large complex systems, MBT has been shown to be an efficiently scalable method.

As a result of industrial adoption of model-based testing (MBT) and test automation in general, the challenges and the main part of manual effort in testing has shifted from test execution to test design and analysis of the results. This has also changed and increased the required expertise of test engineers.

With modern test generation tools used in model-based testing (MBT) it is fairly easy to generate so large amount of test cases that even fully automated execution takes too long time to fit into modern agile development processes. Therefore generated test suites should to be automatically optimized by selecting the test cases that best satisfy the selected testing goals. Chapter 2.4 presents the approaches for evaluating and optimizing test suites.

2.1. Domain Specific Modelling for MBT

As the systems built in the industry grow in complexity and size of the software, it gets more and more challenging to manage and test the systems. The usual way for abstracting the complexity of the system is through modelling. Even when using general purpose modelling languages, the way of using the language and possible extensions (for example UML and its profiles) to model real industrial systems is in practice specific to the system or the domain of the system. Therefore it is getting more common to use domain-specific modelling languages (DSML) to make the modelling easier and to avoid the overhead coming from the use of general purpose languages.

There are varying reasons for the complexity of the system. The system can be large, distributed, configurable, or it can consist of many subsystems or different platforms and devices. Since one model of the system would grow to be too large to comprehend and handle, the usual way is to divide the system into sub-systems that are defined into separate models of varying level of abstraction. Dividing the models into sub-models makes it more challenging to automatically generate executable tests from the separate models. A model used in test automation should include also inputs and expected outputs (test oracles). In complex systems the amount of possible inputs can be enormous, and therefore it should be possible to generate or record the test data, as well as manually add the most relevant data.

In the context of software testing, we can generally see various levels of domain-specific modelling [Kanstren2013]. At the basic level, scripting languages are commonly written to contain language elements from the domain, and parameterized into what can be described as textual domain-specific languages for testing. Means to combine this with model-based testing and to generate such languages from the test models have been described in [Kanstren2012b].

Graphical domain-specific languages can be created with tools such as MetaEdit+. These can be used to build domain-specific test languages such as presented in [Puolitaival2011] and extended to model-based testing in [Kanstren2012a].

Using a suitable existing DSML or specifying a new DSML makes it possible to set the level of abstraction so high that a non-technical domain expert is able to use it to model systems or the expected behaviour of systems. When specifying a new DSML, a suitable generic meta-model can be re-used as template and re-parameterized for the current test instance. This kind of high-level modelling can be exploited also in test automation. The technical system expert can provide a set of "testing building blocks" (e.g., partial testing models or test script fragments) to the domain expert in a form that allows the domain expert to easily construct a test suite using these building blocks. One suitable form could be a domain-specific modelling (DSM) tool describing how the building blocks can be combined into realistic tests.

Interesting future work remains in providing for more advanced support for specific domains, such as VoIP in telecommunications. The more specific these can be made, the easier it usually is for the end user (domain expert) to apply them, and the more powerful they can be. However, it is commonly a difficult goal to combine the required software engineering, domain modelling, and test modelling skills. The domain of protocol testing can be seen as potentially one of the most fruitful areas for this, as once an expert creates a test model, and combines it with suitable test generators, algorithms and test executors, these can be applied across the protocol implementations with high efficiency. An example of such approach for the session initiation protocol is described in [Kanstren2014c].

2.2. Model-Based GUI Testing (MBGT)

Model-based GUI testing (MBGT) aims to automate and systemize the GUI testing process [Grilo2010]. There have been various attempts to develop models to automate some aspects of GUI testing. Although there are approaches that use general purpose modelling languages, such as UML, to model GUI application, a majority uses languages that are specific to the domain of GUI applications. The most popular amongst them are state-machine models that have been proposed to generate test cases for GUIs. The key idea of using these models is that a test designer represents a GUI's behaviour as a state machine; each input event may trigger an abstract state transition in the machine. A path, i.e. sequence of edges followed during transitions, in the state machine represents a test case. The state machine's abstract states may be used to verify the GUI's concrete state during test case execution [Memon2007].

Paiva developed an add-in to Spec Explorer, a model based testing tool developed by Microsoft research, in order to adapt it for GUI testing [Paiva2005]. She has developed a tool to map model actions with real actions over GUI controls. When the mapping information is defined for every model action, the tool generates automatically code that is needed to simulate user actions on the GUI for test execution. Conceptually, during test execution, related actions (model actions and concrete actions) run at the same time and after that results obtained are compared. Every time there is an inconsistency, it is reported.

[Cunha2010] proposes a pattern-based approach and PETTOOL for semi-automated GUI testing of web-based applications. User has to guide the tool during the modelling and testing. The approach identifies patterns of GUI behaviour and provides a generic solution for GUI testing. The proof-of-concept approach has been tested against web-based applications, such as Gmail.

2.3. Model-based GUI testing of smartphone applications

Smartphone applications are in several ways a challenging target for MBT. To begin with, they present the ordinary difficulties inherent in model-based GUI testing. Especially problematic is the fact that during the development the GUI tends to be quite volatile, which makes keeping the models up to date difficult. Design and maintenance workload is further increased by the fact that turnover on the market is typically very rapid. It is not uncommon to have multiple new products constantly under development. However, these products are often quite similar to each other.

Another difficulty is the amount of concurrency involved. Modern smartphones have numerous different applications, from the mainstays like Messaging and Music Player to the more individual user-installed applications. The interactions between these applications can be highly complex as they communicate with each other (e.g. Messaging picking a recipient for a message from Contacts) or compete for hardware resources (e.g. both Gallery and Music Player seeking access to the speakers).

One method to solving the issues of MBT in this domain is dividing the model into multiple components [Kervinen2005]. Instead of one monolithic test model which can be used to test a specific combination of applications on a specific product configuration, one creates multiple smaller model components, each modelling a specific aspect of the system under test (SUT). These components can then be combined into a single test model for test generation. Apart from making individual models easier to understand, this approach enables variation in the test model, as the specific components that compose it can be selected as needed.

A more advanced version of the compositional method is a model library [Jääskeläinen2011], a collection of model components along with semantic meta-information that identifies their role in a test model and applicability to different products and configurations. Individual components in the library can be used for all SUT:s for which they are applicable, thereby reducing redundant design and maintenance work.

There are some specific approaches that have been found helpful in building model libraries. First, testing concurrent systems can be facilitated by dividing the test model into components along the lines of the concurrent units in the SUT, such as modelling Messaging and Contacts as separate (collections of) components. Shared resources can be subjected to the same treatment: if the number of contacts stored on the phone is modelled in a separate component, then all the applications depending on it can access it independently of each other. To enable proper concurrency testing the model formalism must keep track of the state of each component separately, though on a smartphone where only a single application is active at a time their states can be updated individually. Furthermore, some mechanism to facilitate switches between applications is needed, as modelling them manually for all possible combinations of states is in practice infeasible.

Second, further division of models into components within applications can not only make models simpler and easier to handle, but also improve their reusability on different products. For example, one version of Messaging might support distinct audio messages, while another lacks such a feature. If audio messages are modelled as a separate component, they can be included into the test models built for products that support them, even though the rest of the Messaging models may be shared between both kinds of products. With the meta-information in the model library, the assembly of the test model for different products can be automated easily enough, and thus causes no extra work for the testers.

Third, both reusability and maintainability of the models can be improved by designing separate models for functionality to be tested and its implementation on the GUI. This is important, because while different versions of the same product and even entirely different products often support the same functionality, such as basic message sending and reception in Messaging, their GUIs are rarely identical. When the two concerns are separated the abstract functional models can be more easily reused on multiple products, and require less maintenance since they are indifferent to changes in the GUI. The implementation models may need to be revised for new versions and products, but their creation is generally a far more straightforward task compared to functional modelling.

Fourth, when it comes to testing devices with similar control schemes, such as touch screens, further improvements can be made with the use of suitable test execution frameworks. The framework can provide a unified interface based on the concepts of the control scheme and implement them by whatever means suit a specific product. For example, the interface for touch screen devices could provide methods for different gestures such as tap and swipe, and a particular product might be controlled through a specific testing interface or even a physical robot. The primary benefit of abstracting the details of the control scheme into the framework interface is to greatly improve the reusability of the models. In addition it makes models easier to understand and maintain, since they can be based on familiar high-level concepts. Existing high-level test execution frameworks include Selenium WebDriver, which allows testing in different browser environments, and Appium, which uses Selenium to test smartphone applications. Another one currently under development is OTRP, which is designed to test touchscreen devices.

2.4. Automatically Extracting Models for Testing

Crafting models for effective MBT requires a great deal of expertise in formal modelling and a deep understanding of the problem domain. Constructing the models manually from the scratch requires also a significant amount of effort [Grilo2010]. The effort and expertise required for manually crafting the models for MBT is a major obstacle slowing down its industrial adoption. For existing systems, model extraction (also referred to as model inference, specification mining, and reverse engineering) can be used to automate some part or even the whole process of creating the models. Generating models for testing can target either general system properties or specific domain-specific properties.

Machine-learning techniques, such as [Bowring2004, Haran2007, Jin2008], build models that classify a system into categories such as “pass” or “fail” in the context of software testing. Typically such techniques focus on building models based on low-level information such as program points covered and branches taken through the code.

For modelling higher-level software behaviour the basic approaches start from describing the generic control- and data-flow aspects of the system. For example, Daikon [Ernst2000] is a tool intended for providing invariants over observed program traces. That is, it looks at a set of recorded executions over chosen program points and provides invariants describing the observed variable data value at those points such as X always greater than 0. At the simplest level, this can be used as input in software testing to provide information when the software changes and these invariants become violated in the changes [Ernst2000].

Combining invariants with control-flow observations brings the produced models again slightly higher in the level of complexity and abstraction that can be described. For example, the data invariants can be combined with observations of what paths they have been associated to as a way to produce compatibility tests for assessing potential component compatibility [Mariani2007]. Such models can further be applied in test automation by forming state-machines that generalize the observed flows. These can be used e.g. for optimizing a test suite by choosing which set of test cases to execute in order to achieve desired coverage of this model [Lorenzoli2008].

GUI Models

State-based models, such as Finite State Machine (FSM), have been commonly used for modelling Graphical User Interfaces. Miao et al. [Miao2011] propose a finite-state machine (FSM) based GUI Test Automation Model (GUITAM). In GUITAM, a state of the GUI is modelled as a set of opened windows, GUI objects (widgets) of each window, properties of each object, and values of the properties. Events or GUI actions performed on the GUI may lead to state transitions and a transition in GUITAM is modelled with the starting state, the event or GUI action performed, and the resulting state. To reduce the amount of states into computationally feasible level, not all different property values are considered for distinguishing different states of GUITAM.

Aho et al. [Aho2011_ITNG] use a state based GUI model to capture the behavior of the GUI application into state machine, and hierarchical tree models to capture the structure and context of the GUI in each state. The structural models, as well as screenshots of the GUI, are automatically mapped to the corresponding states in the behavioral GUI state model. Events or GUI actions are modeled as transitions between the states, and part of the context information is captured into transitions of the behavioral GUI state model. To avoid the state space explosion, Aho et al. [Aho2011_ITNG] disregard data values, such as texts in text fields or selected values of lists or drop-down boxes, when distinguishing new GUI states from the already visited states. Instead, the data values of the GUI are captured into properties of GUI actions or events. Therefore, the generated GUI state models may have usually an infinite number of transitions, but a reasonable number of GUI states.

Paiva [Paiva2003] used specification language based on ISO/VDM-SL standard to model the UI. Then they started writing the formal models in Spec# and using Microsoft Spec Explorer tool to convert the models into finite state machines (FSMs) [Paiva2005_ASM]. Then the FSMs are converted to Hierarchical Finite State Machines (HFSMs) to reduce the number of states in models.

Memon et al. propose event flow graph (EFG) to model the behavior of the GUI [Memon2001_PhD], based on events instead of GUI states. In EFG, each node represents an event, and all events that can be executed immediately after this event are connected with directed edges from it [Memon2001_PhD]. The EFG has evolved to a more compact GUI model called event interaction graph (EIG) [Xie2005] that can be automatically transformed from an EFG. EIG includes only system-interaction events and with the smaller set of events it better suits rapid testing, such as smoke regression testing. EIG was integrated also into DART to obtain smoke test cases for GUIs to be used for stabilizing daily software builds [Memon2005_IEEE]. After EFG and EIG models, Memon and his team have introduced event-based models, e.g., Event-Flow Model [Memon2004_OOPSLA], [Memon2007], event-space exploration strategies (ESES) [Memon2007], and Event Semantic Interaction (ESI) relation modeled as a graph called the ESI Graph (ESIG) [Yuan2007_ASE], [Yuan2010_IEEE].

Arlt et al. have proposed event dependency graph (EDG) [Arlt2012_CoRR], [Arlt2012_ISSRE] to capture more information to the created models or make the modeling or test case generation based on the models more efficient. In the recent research they have combined static analysis to improve the dynamic GUI ripping and testing process [Arlt2012_ISSRE].

Automated Extraction of GUI Models
In the area of graphical user interface (GUI) software, there are a few approaches using static analysis of the source code for automatically constructing GUI models, such as [Silva2010]. Campos and his research group have numerous publications on static reverse engineering of GUI models [SilvaJC2006], [SilvaJC2006_ATEM], GUIsurfer tool [SilvaJC2009], [SilvaCE2012_EICS] and using the generated models for usability analysis and GUI testing.

The dynamic approaches that involve executing the GUI application and observing the application during the run-time are better suited for extracting the behaviour of GUI applications [Grilo2010]. Memon and his research group have extensively published their research results on GUITAR GUI testing framework [GUITAR], [Nguyen2014] that has been the main platform for their research on automated GUI testing. Their approach for reverse engineering GUI models is called GUI ripping [Memon2003_WCRE]. GUI ripping is a dynamic process for automatically analyzing the structure of GUI [Memon2003_SM] and using the captured GUI structure to create event-flow graphs and an integration tree. GUI ripping was introduced with a framework called DART (Daily Automated Regression Tester) [Memon2005_SME] that automates everything from structural GUI analysis, test case generation, test oracle creation, to code instrumentation, test execution, coverage evaluation, regeneration of test cases, and their re-execution. The goal has been to develop fully automated modeling and testing approaches.

The GUI Ripper was initially implemented for Java SWT-based GUIs, but it has been extended to support Web-based GUIs, iOS, Android, Java JFC, Eclipse and UNO (Open Office) frameworks. The testing process with GUITAR consists of 4 main steps [GUITAR]:

1. GUI ripping: Using a crawler-like tool called GUI Ripper to automatically launch and execute the application under testing (AUT). The GUI Ripper tries to expand hidden GUI elements and all the possible GUI windows, and capture the structure of the GUI into an XML-based model called GUI Tree (or GUI Forest). Each node of the GUI Tree represents a window and encapsulates all the widgets, properties and values in that window [Memon2003].
2. Model construction: Using gui2efg or another model converter to construct an event-flow graph (EFG) or another event-based graph model from the GUI Tree. In EFG, each node represents an event, and all events that can be executed immediately after this event are connected with directed edges from it [Memon2003].
3. Test case generation: Using graph traversal algorithms to generate test cases or event sequences by walking on the event-based graph model with a given coverage criteria, such as covering all events or all edges.
4. Replaying: Using Replayer tool to execute test cases and verify the results.

Various tools have been integrated into the GUITAR framework, such as a profiler tool to dynamically capture actual event-level software usage information while the GUI is being used in the field and employ the information to support refactoring [Nagarajan2003] or to test GUI components [Memon2006], [Brooks2007]. GUITAR has been extended also to support automated GUI testing in open source software projects [Xie2006_ICSM] and agile development processes [Memon2007_agile], and iteratively improve GUI test cases by run-time feedback from executed test cases [Yuan2007_ICSE], [Yuan2008], [Yuan2009], [Yuan2010_IST] or a genetic algorithm to evolve new test cases that increase test suite's coverage while avoiding infeasible sequences [HuangS2010]. GUITAR and GUI ripping has been used also for evaluating and improving usability of GUI applications [Memon2009].

Memon's team has extensively published their research results also in the areas of coverage and effectiveness of GUI testing [Memon2001_ESEC], [Xie2006_ICSE], [Xie2006_ISSRE], [Strecker2007], [Strecker2008], [Xie2008], [Yuan2011], [Strecker2012], and test suite reduction and prioritization for automated GUI testing [McMaster2005], [McMaster2006], [Bryce2007], [McMaster2007], [McMaster2008], [Brooks2009_ICSM], [Elsaka2010], as well as characteristics of industrial GUI systems [Brooks2009_ICST].

Ana Paiva and her team in University of Porto (Porto, Portugal) have extensively researched GUI modelling and testing. They reverse engineered formal GUI models using semi-automated dynamic technique, mixing manual and automatic exploration to access parts of the GUI that are protected by a key or are in some other way difficult to access automatically [Paiva2007_FMICS]. The goal of the approach and presented REGUI2FM tool is to reduce the modeling effort and provide mapping information for executing abstract test cases on a concrete GUI during model-based GUI testing process. The extracted partial as-is Spec# model is validated and completed manually into "should-be" model, including also expected outputs to be used as test oracles. During the exploration process, the intermediate code of the AUT is instrumented with Aspect-Oriented Programming (AOP) techniques in order to be able to recognize and capture a wider range of GUI controls and events, beyond native ones. Spec Explorer tool is used for transforming Spec# model into FSM, and generating and executing test cases. Paiva et al. [Paiva2007_notes], [Moreira2008] have also provided UML based modeling tool

VAN4GUIM for abstracting the complexity and visualizing the textual Spec# models.

With J.L. Silva and Campos [SilvaJL2007] Paiva combined dynamic GUI reverse engineering with manual GUI modeling to improve the test oracles of the created models. In this approach they used ConcurTaskTrees (CTT) task models to describe UI interactions and TERESA tool [Teresa] to generate FSM models from the CTT. They developed TOM (Task to Oracle Mapping) tool to generate Spec# models with test oracles from these FSM models, reverse engineered the mappings between model and concrete GUI objects into another Spec# model, combined the two Spec# models, and then used Spec Explorer to generate a new FSM model that is used for model-based GUI testing.

The reverse engineering approach was further developed with Grilo [Grilo2009] by implementing the dynamic reverse engineering algorithm on top of Microsoft UI Automation library [MsUI] and saving the GUI model in XML based format before generating the Spec# specification for testing purposes, and the tool was renamed to REGUI [Grilo2010]. The approach combines automated and manual exploration to access all parts of the GUI, but the user has to guide the tool during the creation of the model, and the generated model has to be validated and completed manually.

The ReGUI tool was developed further into version 2.0 with Morgado [Morgado2011] using a different, fully automatic reverse engineering approach. The generated models include a structural model of the GUI, saved into XML based ReGUITree model, and behavioral models saved into four GraphML models and one Spec# specification. The approach is extended by generating a Symbolic Model Verification (SMV) model for model checking purposes [Morgado2012_AiS], and generating UML 2.0 FSM model and using Inductive Logic Programming (ILP) machine learning technique to solve ambiguous situations, e.g., when the same event may lead to multiple different target states. Together with R. Ferreira they have researched also test coverage analysis [FerreiraR2010_QUATIC], [FerreiraR2010_ICSTW].

In the recent research Silva et al. [SilvaCE2013] combined dynamic analysis with static source code analysis for reverse engineering Web applications. First, a state based model of the Web UI is obtained using dynamic reverse engineering, and then the relevant conditions over the input values in the UI are determined by static analysis of the event handlers attached to each UI control. An Abstract Syntax Tree (AST) is created for each event listener.

Amalfitano et al. have been recently extremely active in publishing their research on dynamic reverse engineering of rich internet applications (RIAs) and mobile applications. Their approach for reverse engineering client-side behavior of RIAs [Amalfitano2008] is based on dynamic analysis of the RIA execution traces and employing clustering techniques for solving the problem of state explosion of the generated finite state machine (FSM) models. The developed RE-RIA tool creates an instantiation of a Mozilla Firefox Browser inside its Java GUI and the approach is used to model AJAX based Web applications [Amalfitano2009_ICSM].

In the domain of web-applications, models have been generated by crawling through the various possible navigation paths of the system [Mesbah2012]. This can then be combined with user defined invariants over the model (such as information in table should match given observed constraints) when the tool in future re-crawls the same application to produce test cases. These can be further refined using various domain-specific criteria such as web-application navigational elements, exploration history, and GUI structural coverage [Fard2013].

Using Extracted Models for Testing

In general, going from an automatically generated test model to meaningful test cases is a challenge and always only able to address only specific generalized aspects. In going from such models to higher level and more powerful models with test oracles and test data tailored for the system under test, one needs to augment these models. One is to add specific checks for tailored invariants such as done in [Mesbah2012]. In general a procedure for such modelling is described in [Kanstren2010]. In this procedure, one starts with the generated model, augments it with expert domain knowledge, generates test cases, evaluates the results, refines the model again with help of domain experts, and iterates this process [Kanstren2010]. This is an application of specification mining to model-based testing. Further input for this process could be generated by applying various data generation, crawling and other similar approaches to refine the observed invariants and to build a more explicit feedback loop for the domain experts. Some initial work in such direction for web-applications is shown in [Fard2013], although still missing the domain-expert in the loop.

Amafitano et al. used the generated models for testing RIAs [Amalfitano2010_ICSTW] and use test reduction techniques test suites to ensure scalability. Execution traces used for the test case generation can be obtained either from user sessions, automated crawling of the application or by combining both approaches. Combining both manually and automatically obtained execution traces increased the effectiveness of testing. [Amalfitano2010_ICSTW] introduces tools developed for the process: an AJAX crawling tool CrawlRIA, a dynamic RIA analysis tool CreRIA, Test Case Generator and Test Case Reducer tool TestRIA, and DynaRIA [Amalfitano2010_ICPC], [Amalfitano2010_QUATIC], [Amalfitano2013_Springer], a tool for dynamic analysis and testing of RIAs. The approach can be used also in agile processes [Amalfitano2010_ICIW], and CReRIA tool has been extended to support semi-automatic generation of user documentation for Web 2.0 applications [Amalfitano2011_WSE].

Amalfitano et al. [Amalfitano2009_ASEA] have presented a Web page classification technique and WPC-CA (Web Page Classifier) tool based on the deduction of classification rules that allow the reliable classification of the pages of a Web application. A reliable classification of Web pages can be useful for supporting various engineering activities, such as re-engineering, analysis and testing. They have proposed a classification framework that characterizes existing RIA testing techniques [Amalfitano2010_WSE].

In more recent research [Amalfitano2011_ICSTW] they presented a technique and A2T2 (Android Automatic Testing Tool) for dynamic reverse engineering and automated testing of Android applications. The tool has been developed in Java and is composed of three main components: a Java code instrumentation component, the GUI Crawler and the Test Case Generator. It supports rapid crash testing and regression testing of Android applications. Amalfitano wrote also his PhD thesis [Amalfitano2011_PhD] on reverse engineering and testing of RIA and Android applications.

Together with Memon, Amalfitano et al. [Amalfitano2012_ASE] presented AndroidRipper approach for dynamic reverse engineering of GUI models of Android mobile applications. The approach uses a high-level automation library called Robotium [Robotium] and GUI ripper whose behavior can be configured according to application specific testing aims. While dynamically exploring the GUI, the tool detects run-time crashes of the application. The goal of the approach is not to develop a model of the application, but automatically and systematically traverse the GUI, generating and executing test cases as new events are encountered. Test case generation is based on the automatic dynamic analysis of the GUI that is executed in order to find and fire events in the GUI. The approach is further automated

[Amalfitano2012_ICSM] to reduce the manual intervention during the testing process.

Their approach for testing Android mobile applications is further extended [Amalfitano2013_ICSTW] to support a broader set of events. As the earlier focus had been only on user events produced through the GUI, now they considered also context events, e.g., events coming from the external environment, events generated by the device hardware platform, events typical of mobile phones, and events from other Internet connected applications. The proposed testing techniques involve the manual definition of reusable event patterns including context events. Event patterns may be used to manually generate test cases, to mutate existing test cases, and to support the systematic exploration of the behavior of an application using the GUI Ripping technique.

As the generated models are based on the behaviour of the observed implementation, instead of the specifications or expected behaviour, it is challenging to automatically generate meaningful test oracles. In most of the dynamic GUI reverse engineering approaches for testing, the test oracle is based on the observed behaviour of an earlier version of the GUI application. Using this kind of test oracle, changes and inconsistent behaviour of the GUI can be detected, but validation and verification against the specifications is problematic. Although using implementation based models in testing has restrictions and requires special consideration, the generated models can be used in automated testing and supporting various manual testing actions.

Aho et al. present GUI Driver tool. During the automated model extraction, the GUI is dynamically traversed or crawled through by interacting with all the detected widgets of the GUI [Aho2011_ITNG]. As the number of possible event or interaction sequences is enormous, GUI Driver attempts to drive the software into as many different states as possible, aiming for state coverage, instead of transition coverage [Aho2011_ITNG].

Testing of the application begins already during the modeling, as crashes, unhandled exceptions and some usability issues are detected and reported [Aho2011_ITNG]. 3rd party MBT tools are used for generating tests from the models. GUI Driver uses model transformation to save the GUI state model into GraphML, to allow the use of external graph traversal algorithms for generating test sequences with given coverage criteria [Aho2011_ITNG]. The generated test sequences can be executed with GUI Driver and test oracles are based on an earlier, presumably correct version of the GUI application, allowing the regression testing [Aho2011_ITNG].

The fully automated reverse engineering approach [Aho2011_ITNG] was extended into semi-automated iterative process [Aho2011_ICOS] of automated reverse engineering and manually providing valid input, such as username and password, when the reverse engineering algorithm does not find any more new states from the GUI application. The user provides the input combinations into the GUI that is being modeled and then allows the GUI Driver tool to continue automated reverse engineering [Aho2011_ICOS].

Aho et al. [Aho2013_CoDIT] have highlighted the importance of diverse GUI automation and interacting with all enabled widgets of the GUI during the dynamic reverse engineering in order to produce more comprehensive models of the dynamic behavior of GUI applications. Therefore they proposed a widget classification to support GUI automation strategies for effective model extraction [Aho2013_CoDIT] and presented some easy-to-implement strategies to demonstrate the feasibility of the classification. The goal of the GUI automation strategies is to reach as many states as possible with as few events or interactions as possible [Aho2013_CoDIT], in similar way as in test suite prioritization.

During this project, VTT and F-Secure collaborated on Murphy, a dynamic GUI reverse engineering and testing toolset developed as internal tool at F-Secure [Aho2013_EESSMod]. The idea of Murphy tool is similar to GUI Driver: traversing or crawling through all the possible states of the GUI application and automatically constructing a state based GUI model of the behavior observed during the execution. Most of the model extraction approaches have limitations and restrictions on the GUI applications that can be modeled, but Murphy is more platform independent, supporting most GUI platforms regardless of the implementation language [Aho2013_EESSMod]. Murphy provides also a variety of tools to use the extracted GUI models for automated testing and supporting manual testing activities, supports virtualized test execution environment, and integrates to other tools in automated test environment, e.g., Jenkins [Jenkins] open-source continuous integration tool.

Murphy tool [Aho2013_EESSMod] uses state based directed graph for capturing the behavior of the GUI application and models events or interactions as transitions between the states. During the UI crawling, screenshots of the GUI are automatically captured after each interaction and used for visualization of the resulting graph models [Aho2013_EESSMod]. In addition to using the traditional means of dynamically observing the GUI, such as UI Automation library [MsUI], Murphy provides its own window scrapping libraries. One of the libraries innovatively uses image recognition and comparison to provide platform independent way to detect widgets for interaction and changes in the GUI for observing the state changes and possible defects [Aho2013_EESSMod].

As the extracted models are based on the observed implementation, instead of requirements of the system, visual inspection and manual approval of the models is required to make sure that the modeled application behaves as expected [Aho2014_TAIC-PART]. Using screenshots of the modeled GUI application to visualize each state of the model helps in reading the models and understanding the behavior of the modeled application [Aho2014_TAIC-PART]. In addition to detecting crashes and unhandled exceptions during model extraction and testing, test oracles are based on approved behavior of an earlier version of the GUI application [Aho2013_EESSMod]. The extracted models of a new version are compared with the models of the previous version using model comparison functionality, and Murphy reports deviations, showing screenshots of both versions and highlighting the changes in the images. Then the user has to decide which changes were desired features and which are deviations from the expected behavior [Aho2014_TAIC-PART].

One of the main contributions of Aho and Suarez [Aho2013_EESSMod], [Aho2014_TAIC-PART] was presenting experiences from using the Murphy tool to automatically extract models from commercial GUI software products and using the models to automate and support GUI testing activities in highly automated industrial development and testing environment.

3. Test Evaluation and Optimization

With modern test generation methods such as model-based testing, building large test suites becomes quite simple, since test case construction is easily automated. The problem with this is that a large number of test cases do not automatically imply that all these test cases are indeed useful or nec essary. Indeed, it is quite likely that many of the selected test cases are redundant, or that their results can be derived from earlier test phases. Also, automated test case generation does not usually lend itself to guaranteeing a given kind of coverage unless the test case generation method is specially built for this.

Also, redundant generated test cases often remain hidden until test case execution. If we consider the case of model-based testing, the process is started promptly at design, and theref ore enabling early fault detection. Test suites generated with redundancy at test case level can be detected and removed earlier in the testing process. From all testing activities test suite generation is the most crucial part [Bertolino2003]. Evaluating the test suite based on some given criteria, i.e. coverage and fault-based criteria, is demanding and has multiple theoretical aspects. Based on the chosen generation mechanism one can find that evaluating the test suite in a systematic way is not a simple task.

Evaluation and optimization of test suites has a direct impact on the costs and effort of software testing. In order to deal with these issues several questions need to be addressed. First of all, one needs a suitable test suite optimization technique suitable for model-based testing. Current amount of evaluation is insufficient to identify a single superior technique with regards to whatever criterion and therefore it is not necessary to decide on a single technique. In fact it can be the cas e to use a combination of several different techniques.

The process of identification, removal, prioritization of test cases that finally can lend an optimal test suite with regard to some criterion, can be defined as test suite optimization. As redundan cy of a test case can be relative, the optimization techniques need to deal with all possible combinations of the test cases in a test suite. The process of manual optimization is both overwhelmingly complex and not desirable. Also, exhaustive optimization even with an automated technique would give results only in toy example and is impractical for industrial integration. We can assume that the complexity of any test suite optimization technique is exponentially related to the test suite cardinality.

Any type of test suite optimization, minimization or similar activity requires initially one to define what the test suite is being optimized in relation to. Typically this is considered as a form of test coverage. In general software testing, we can see this as code coverage and traditional measures for this include code coverage, path coverage, and decision coverage. Other measures may be necessary in different fields of test automation. For example, model-based testing is commonly applied as a black-box approach and in this case the coverage criteria typically become criteria over the test model. While the test model can be seen as code, the coverage criteria are different from the traditional generic ones for systems under test as there is generally more information in the models tailored for testing purposes only, while at the same time the black-box view excludes the availability of source code for the system under test (or at least require consideration for cases where this is true). Specific coverage crit eria for test models then include structural model coverage (e.g., model states and transitions), data coverage (e.g., model variable values), requirements coverage (represented e.g., as function points in the model), test case specifications (e.g., user defined scenarios over the model paths), random walks (e.g., to de-emphasize pure expert bias), and fault-based (such as history or mutation coverage) [Utting2012].

3.1. Model-checking Tailored Test Generation, Evaluation and Optimization

Model-based testing by model-checking is a technique introduced almost fifteen years ago [Engels1997] as an efficient way of using a model-checker to interpret traces as test cases. More details and references on testing with model-checkers can be found in the work of Fraser et al. [Fraser2009]. Within the last decade model-checking has turned out to be a useful technique for generation of test cases from finite-state models [Engels1997]. However, the main problem in using model-checking for testing industrial software systems is the potential combinatorial explosion of the state space and its limited application to models used in practice.

A model checker has been used to find test cases to various criteria and from programs in a variety of formal languages [Hong2002],[Ammann2002]. In addition, Black et al. [Black2000] discuss the problems encountered in using a model-checker for test case generation for full-predicate coverage and explain why logic coverage criteria is not directly applicable for model-checking. Rayadurgam et al. [Rayadurgam2003] present an alternative method that modifies instead the system model and are obtaining MC/DC adequate test cases using a model-checking approach.

Using Logic Coverage to Improve Test Generation and Evaluation
In model-driven development, testers are often focusing on functional model-level testing, enabling verification of design models against their specifications. In addition, in safety-critical software development, testers are required to show that tests cover the structure of the implementation. Testing cost and time savings could be achieved if the process of deriving test cases for logic coverage is automated and provided test cases are ready to be executed. The logic coverage artifacts, i.e., predicates and clauses, are required for different logic coverage, e.g., MC/DC. One way of dealing with test case generation for ensuring logic coverage is to approach it as a model-checking problem, such that model-checking tools automatically create test cases. There have been a number of testing techniques used for defining logic coverage using model-checkers, e.g., [Black2000, Rayadurgam2003, Rayadurgam2001].

Logic-based Coverage Criteria
In the literature, there are many similar criteria defined, but with different terminology [AmmannOffutt]. Also, some definitions of coverage criteria (e.g., MC/DC) have some ambiguities. In order to eliminate the ambiguities and conflicting terminologies, Ammann et al. [Ammann2003] abstracted logic criteria with a precise definition and formal representation. A predicate is an expression that evaluates to a Boolean value. It consists of one or more clauses. A clause is a predicate that does not contain any logical operators and can be a Boolean variable, non-Boolean variables used for comparison, or a call to a Boolean function. Clauses and predicates are used to introduce a variety of coverage criteria. This paper presents three different test criteria, each of which requires a different amount of test cases:

(1) Predicate Coverage (PC)
(2) Clause Coverage (CC)
(3) Correlated Active Clause Coverage (CACC).

These are defined in the next sections in terms of the FBD program. We note that CACC relies on its original definition and it is similar to that of modified condition/decision coverage (MC/DC).

Automated Test Generation

In this project, our goal is to help testers automatically develop tests for safety critical software systems modeled in Function Block Diagram. One example includes logic coverage, which needs to be demonstrated on the developed programs. There has been little research on using logic coverage criteria for Function Block Diagram programs in an industrial setting. In some cases logic coverage is analyzed at the code level. Even if at the code level, logic coverage is used, it would be difficult to standardize the code generation scheme for different PLC tool vendors in order to map directly the criteria to the original Function Block Diagram program. Hence, in this model-driven environment it is advantageous to move as much testing activity from code level to Function Block Diagram program level as possible.

We developed a framework suitable for transforming Function Block Diagram programs to a formal representation of both its functional and timing behavior. For this, we implement an automatic model–to–model transformation to timed automata, a well-known model introduced by Alur and Dill. The choice of timed automata as the target language is motivated primarily by its formal semantics and tool support for simulation and model checking. Our goal is not to solve all testing issues (e.g., robustness, schedulability, etc.), but to allow the usage of a framework for formal reasoning about testing Function Block Diagram programs. The transformation accurately reflects the data-flow characteristics of the Function Block Diagram language by constructing a complete behavioral model, which assumes a read-execute-write program semantics. The translation method consists of four separate steps. The first three steps involve mapping all the interface elements and the existing timing annotations. The latter step produces a formal behavior for every standard component in the Function Block Diagram program. These steps are independent of timed automata and thus are generic in the sense that they could also be used when translating an Function Block Diagram program to another target language. This allowed us to investigate further test case generation techniques based on model checking.

We developed a testing technique based on model checking, tailored for logic coverage of Function Block Diagram programs. There have been a number of testing techniques used for defining logic coverage using model-checkers. However, these techniques are not directly applicable to Function Block Diagram programs and semantics. Our main goal with this project was to define logic coverage for Function Block Diagram programs based on the transformed timed automata model. This copes with both functional and timing behavior of a Function Block Diagram program. We also found that a formal definition is necessary for the approach to be applicable to model checking. We show how a model-checker can be used to generate test cases for covering a Function Block Diagram program.

We developed a testing tool for safety critical applications and applied it on a large-scale case study. We applied the tool in a large case study and found that it is efficient in terms of time required to generate tests that satisfy logic coverage and scales well for most of the programs.

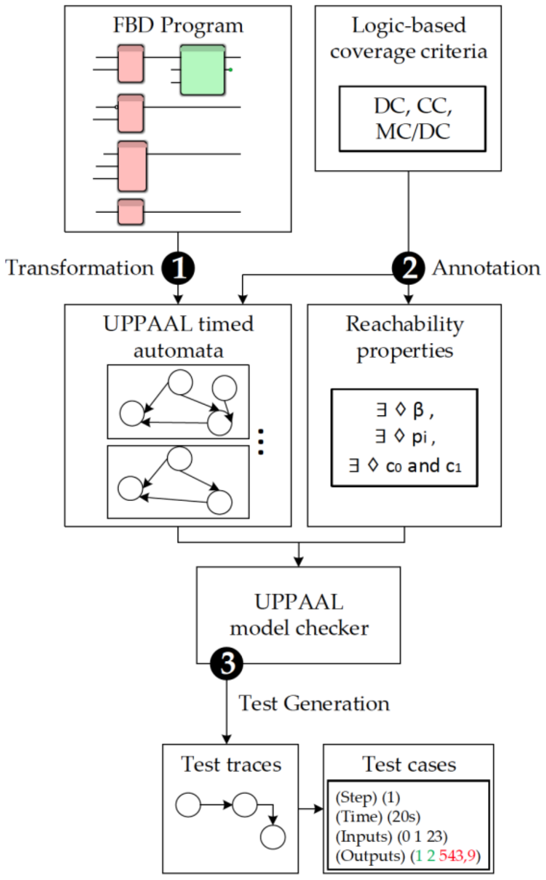

Figure 3.1.1. Testing Methodology Roadmap

In this report, we describe an approach to automatically generating tests for FBD programs. Logic coverage criteria are used to define what test cases are needed and we use a model checker to generate test traces. In addition, the methodology presented in this report is tailored for FBD programs, and is composed of the following steps, mirrored in the figure above:

1. Model Transformation: To test an FBD program we map it to a finite state system suitable for model checking. In order to cope with timing constraints we have chosen to map FBD programs to timed automata.
2. Logic Coverage Annotation: We annotate the transformed model such that a condition describing a single test case can be formulated. This is a property expressible as a reachability property used in most model checkers.
3. Test Case Generation: We now use the model-checker to generate test traces. To provide a good level of practicality to our work, we use a specific model-checker called UPPAAL which uses timed automata as the input modeling language. The verification language supports reachability properties. In order to generate test cases for logic coverage of FBD programs using UPPAAL, we make use of UPPAAL's ability to generate test traces witnessing a submitted reachability property. Currently UPPAAL supports three options for diagnostic trace generation: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.

While UPPAAL is a viable tool for model checking, it is not directly tailored to test case generation in practice. We demonstrate how to work around this by automatically generating traces for logic coverage of the control flow of FBD programs described in timed automata and how we transform these traces to actual test cases.

The basic approach to generating test cases for logic coverage using model-checking is to define a test as a finite execution path. By characterizing a logic coverage criterion as a temporal logic property, model-checking can be used to produce a path for the test obligation. By using a translated FBD program, we use logic coverage to directly annotate both the model and the temporal logic property to be checked. We propose the annotation with auxiliary data variables and transitions in such a way that a set of paths can be used as a finite test sequence. In addition, we propose to describe the temporal logic properties as logic expressions satisfying certain logic coverage criteria. Informally, our approach is based on the idea that to get logic coverage of a specific program, it would be sufficient to **(i)** annotate the conditions and decisions in the FBD program, **(ii)** formulate a reachability property for logic coverage, and **(iii)** find a path from the initial state to the end of the FBD program. To apply the criteria, necessary properties for the integration of logic coverage need to be fulfilled.

Experimental Evaluation of Automated Test Generation

Our goal in this report is to evaluate the automated test generation technique on industrial FBD programs and to acquire experience regarding its efficiency and usability. We therefore conduct a set of analyses using programs developed by Bombardier Transportation AB in Sweden. The system has been in development for more than two years and uses processes influenced by safety critical requirements and regulations including the EN 50128 Standard, which requires different logic coverage levels (e.g., DC and MC/DC). The industrial system studied in this paper is the TCMS (Train Control and Management System), developed by Bombardier Transportation AB engineers, which has been deployed to the field. In this research we, have used all TCMS programs written in the FBD standard language resulting in a total of 157 artifacts. Each of the programs is sizable and representative of industrial programs used in the train system's development.

We investigate the following questions regarding the method's performance:

- Q1, Efficiency: What is the time required for the tool to generate tests that satisfy the DC, CC and MC/DC logic coverage criteria?
- Q2, Coverage: How close does the tool come to generating tests that achieve 100% coverage of each of the criteria?

To answer Q1 and Q2, the tool generates tests aimed at achieving maximum logic coverage. Since we are using a model checker for generating tests, the toolbox simply produces the maximum achievable coverage with a proof that uncovered test obligations are not coverable. For 123 of the 157 programs (78%) the tool provided tests that covered 100% of the required entities for each of the three coverage criteria. The generation time for MC/DC averaged approximately twice as long as for DC. The results are summarized as boxplots in the figure below with the kernel density distribution of the generation time shown also below. The kernel densities estimates for the generation time for DC (red), CC (green) and MC/DC (blue) are plotted on the same graph. It is quite clear on the graph that the distribution of generation times is more variable for MC/DC. It is also worth noting that the generation time modes (i.e., most frequent values in the generation time data set) of a distribution are close to each other for all criteria. We can observe that a few outliers caused the average generation time to greatly exceed the median generation time for all coverage criteria. For 34 of the 157 programs, the tool did not terminate after running for a substantial period of time. After discussions with engineers from Bombardier Transportation AB regarding the needed time for a tester to provide a set of tests for a desired coverage, we concluded that 10 minutes was a reasonable cut-off point for the model checker to terminate.

For 78% of the programs in this report, the tool automatically generated tests achieving 100% DC, CC and MC/DC. For the other 22% of the programs, the results were less satisfactory. As can be seen from the data,

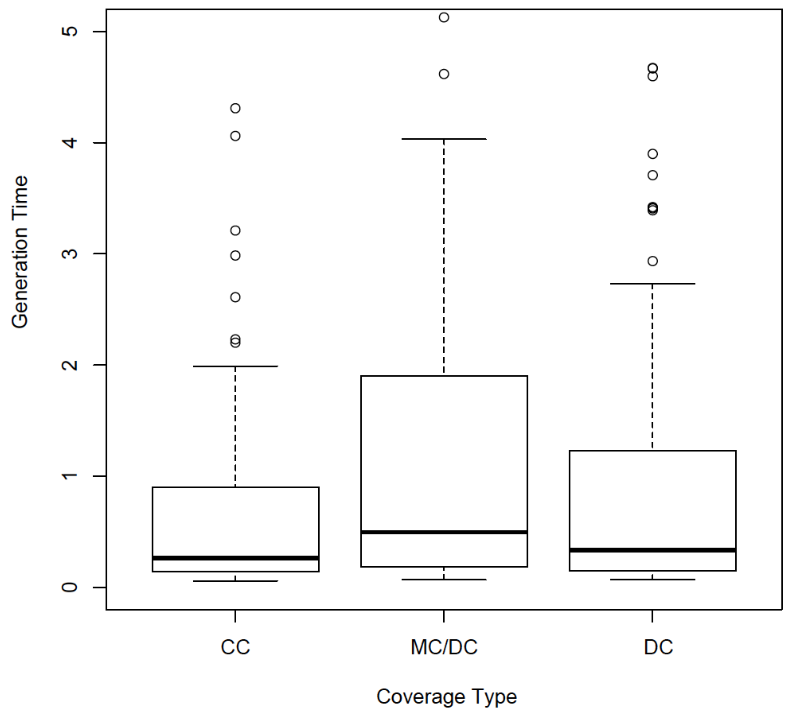

Figure 3.1.2. Experimental results: Generation Time Distributions

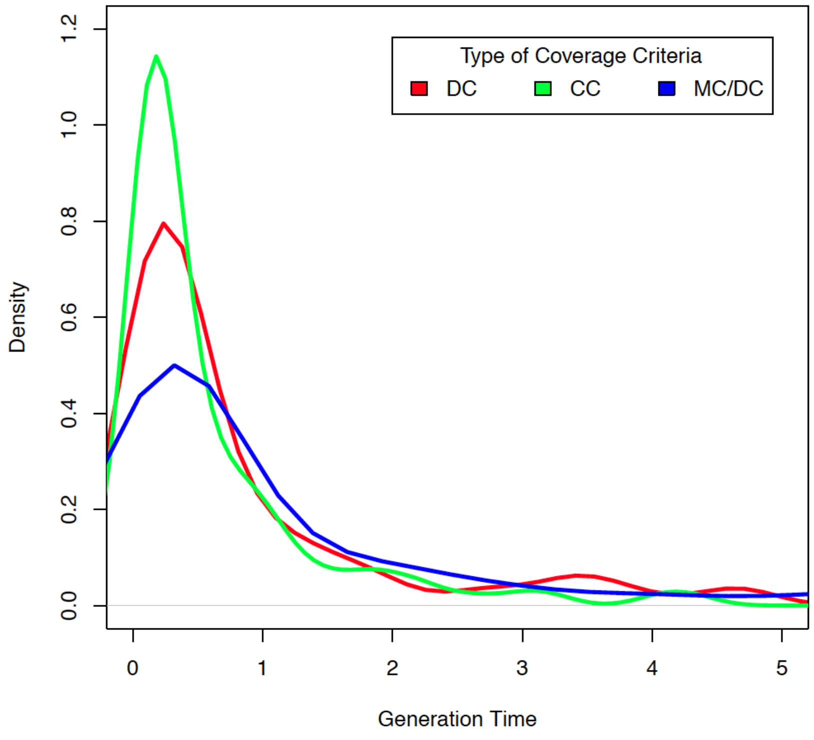

Figure 3.1.3. Generation Time Distribution by Coverage Criteria.

the tool generated tests with 82% DC on average. We conclude that we have provided evidence that this is a suitable tool for test generation tailored to FBD programs; it scaled well for most of the programs in this report and it is fully automated.

There are, however, some drawbacks. Most importantly, for 22% of the programs, even though the tests generated for the coverage criteria achieved on average at least 65% coverage, we cannot determine whether the remaining test requirements are actually achievable, or if tests satisfying the requirements are longer that the search depth. This is an issue particularly for MC/DC where a fair number of test obligations were not satisfied. From these experiments, it is clear that the toolbox can be sensitive to the number of decisions and as a consequence to the length of the tests required to achieve the desired coverage.

3.2. Test Suite Minimization

Test suite minimization implies the use of methods to reduce the size of a given test suite such that a given coverage criterion is satisfied. A test suite that contains a large number of redundant test cases can be considered inefficient. A lot of research effort was put into tackle this problem. Formally Rothermel et al. [Rothermel2002] defined test suite minimization as follows: a test suite T, a set of test requirements Rs, that must be satisfied in order to provide adequate testing of the system, with the problem of finding a representative set T' of test cases from T that satisfies all Rs.

Previous work has been done on test case minimization with regard to different heuristics techniques for the minimal hitting set problem [Chen1996, Offutt1995]. Others, like Horgan and London applied linear programming to test case minimization problem [Horgan1991]. Harrold, Gupta and Soffa described a heuristic based technique at code level in order to remove redundant test cases from an initial test suite [Gupta1993]. Chen and Lau [Chen 2003] described a minimization approach based on divide-and-conquer method that uses a random technique. Xie et al. [Xie2004] described a method for optimization of object oriented unit tests by eliminating redundant test cases. Jeffrey and Gupta formulated a technique for minimizing a test suite with selective redundant test cases [Jeffrey2005]. Bertolino described test suite minimization as a problem of finding a spanning set over a graph [Bertolino2003]. The representation of the system under test used by Bertolino is described as a decision-to-decision graph (ddgraph). The results of data-flow analysis are used into ddgraph for requirements testing, and therefore the test suite minimization can be regarded as the problem of finding the minimal spanning set. Except of the research done on test suite minimization techniques with regard to some coverage criteria, there are other approaches. Harder et al. are using operation abstraction as the formal model of the dynamical system behavior [Harder2003].

Other work has focused on model-based test suite minimization [Vaysburg2002, Korel2002, Black2001, Hong2003, Heimdahl2004]. Vaysburg et al. described a minimization method for model-based test suites that uses dependence analysis of Extended Finite State Machines (EFSMs) [Vaysburg2002]. By using dependence analysis, testing transitions in the model can be seen as testing the set of dependent transitions. With this method it is considered to be a redundant test case, the one that contains the same set of transitions of some other test case. Korel exploited test suite minimization by using this technique in combination with an automatic way of dealing with changes in the models [Korel2002] that is very useful in model-based regression testing. Therefore, test cases with modified transitions are optimized with the dependence analysis-based minimization techniques. Others like Black and Ranville [Black2001] introduced several methods to shrink the size of test suites, such as removal of redundant test cases using model checkers. The problem of finding the minimal subset of test cases is NP-hard [Hong2003]. Using model checkers for minimization is briefly considered in the work of Zeng et al [Zeng2007]. Nevertheless, removing test cases from a test suite has been shown to affect the overall fault detection capability [Heimdahl2004].

3.3. Test Suite Selection

Following the formal definition of Rothermel and Harrold [Rothermel1996], the test suite selection problem is defined as follows: The model, M, the modified version of M, M' and a test suite, T, with the problem of finding a subset of T, T', with which to test M'. This problem fits to test suite optimization problem in the context of regression testing. In the literature the test suite selection techniques are aimed at also reducing the size of a test suite, as the test suite minimization. The majority of the techniques we have looked on are focused on regression testing. Therefore, the test suite selection is not specific to the current version of the system under test, and is focused on the identification of the modified parts. As a direct consequence test suites are optimized with regard to selection of changes in the system under test.

Based on Rothermel's formal definition [Rothermel1996], various approaches are focusing on identifying test case specific to modified parts of the system under test. The specific techniques differ according from one method used to another in terms of definition and identification of the modified parts. Substantial research results have been reported in the literature that are using different techniques and criteria including integer programming [Fischer1977, Fischer1981], data-flow analysis [Gupta1992, Harrold1989], symbolic execution [Yau1987], CFG graph-walking [Rothermel1994], or SDG slicing [Bates1993]. Nevertheless, some studies including [Grindal2006] are reporting a gap between test suite selection techniques and their deployment in industry, which is substantiated by the manual test selection and expert knowledge used nowadays in software industries.

3.4. Test Suite Prioritization

An optimization technique used to improve a given test goal (e.g. fault detection capability) is the test suite prioritization [Rothermel1999], which is used for ordering test cases of a given test suite for early maximization of some desirable properties. It can be defined as the optimal permutation of a certain set of test cases. Following the definition of Rothermel [Rothermel1999] it assumes that all the initial test cases of a test suite may be executed in the produced permuted order with the mention that the testing process during prioritization can be finished at any arbitrary time. Test case prioritization concerns ordering test cases for early maximization of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process. Therefore, test suite prioritization can be used to maximize the optimal path and time in which the given test goal is reached.

One of the most used metrics/criterion in test suite prioritization is the structural coverage [Rothermel1999, Elbaum2001] with the evident goal of maximizing fault detection by maximizing structural coverage. Rothermel et al. studied several test suite prioritization techniques [Rothermel1999] by using the same algorithm with different fault-detection rates (e.g. branch, statement, fault explosion potential). Leon and Podgurski introduced test suite prioritization based on the distribution of profiles of test cases in a multi-dimensional space [Podgurski2003]. This technique aim is to use test profiles resulted from applying the dissimilarity metric in order to determine the degree of dissimilarity between two input profiles. Kim and Porter described a history-based technique for test suite prioritization of test cases [Kim2002]. The main achievement is the usage of test suite selection in combination with test suite prioritization. If the test suite is too large, then the test suite are prioritized until is sufficient.

Other work was directed on model-based test suite prioritization, and was introduced by Korel et al [Korel2005, Korel2007], with the initial goal of using this technique with test suite selection [Korel2005]. Initially the test suite prioritization is done randomly. The test cases were divided into two categories, a high priority set and a low priority set. A test case is assigned into a category based on their relevance to the modification made to the system under test. Usually the test suite optimization starts with the analysis of each test case with regard to its coverage criterion and ends with an arrangement in decreasing order according to their coverage value. Others like Fraser and Wotawa described a model-based prioritization approach [Fraser2007a] based on the notion of property relevance [Fraser2006]. This relevance is defined in the context of using a model checker to a model property and is defined as the test case capability to violate this property. They showed that test suite prioritization based on a model property could be advantageous to be used in comparison with test suite prioritization based on coverage. It should be noted that this advantage has a direct relation with how the model is specified.

3.5. Optimal Test Suite Generation

Optimal test suite generation refers to test suite generation with respect to optimality in terms of ordering and checking the test goals during the test case generation and execution. In comparison to test suite minimization techniques, optimal test suite generation tries to overcome the problem of assuming that all test cases are generated accordingly to a specific technique, and as a consequence to optimize only the number of test cases that have to be executed. If we take into consideration complex systems used in software industry, then it could be beneficially to optimize test suites at generation. This technique is becoming very popular as a research area, especially in model-based optimal test suite generation. Hyoung et al [Hyoung2005], Zeng et al. [Zeng2007], and Fraser and Wotawa [Fraser2007b] have done work on optimal test suite generation with regard to both coverage- and mutation-based approaches. They identified the influence of a model-checker when called for test suite generation. Usually, the model checkers is used for each test property, and as a consequence too many redundant or subsumed test cases are generated [Fraser2007b]. Therefore, when generating test suites with a model checker, the order in which test properties are ordered and selected has a direct relation with the size of the resulting test suite. Hong and Ural [Hyoung2005] are using a model checker to define subsumption relations between test properties described by a coverage criterion in order to reduce the cost of test suite generation.

Researchers are describing test case generation from abstract models by casting the test case generation problem as a model-checking problem. While this is the main focus of research on testing with model checkers, other applications have been considered in the past. Therefore, model-checking techniques have been used in other ways to derive test cases. In contrast to this work, Hessel et al. [Hessel2004] are proposing time optimal test suite generation for timed automata models using UPPAAL model-checker [Larsen1997]. In this approach a variant of CTL, named Timed-CTL, is used to formalize test purposes or coverage criteria. The generation of test suites with either test purposes or properties created for coverage criteria is of particular interest for systems modeled with time, because the method supports optimal test suite generation of not only shortest but also quickest traces.

Similar techniques are also widely applied in offline model-based test generation. The techniques commonly use static analysis based techniques such as symbolic execution and constraint solving on the test models to generate test cases optimized for specific purposes. Examples of this includes Spec Explorer [Veanes2008] and above mentioned Uppaal [Mikucionis2004]. In the context of online MBT generation faster response times are required with less resources to perform analysis, which commonly leads to applying various randomization based approaches augmented with heuristics such as weights and scenarios [Veanes2005]. Combinations of these include using input from separate offline analysis as input for online testing [Ahman2012].

These optimization approaches for MBT typically target specific coverage targets in the test model, such as manually tagged coverage requirements. As one of the key benefits of automated test generation is the ability of a test generator to create and execute large scale variation, more complex coverage and optimization criteria are useful to make better use of such tools. [Kanstren2014a] presents a coverage criteria for covering general variation over the test model, as well as integrating user-defined domain-specific coverage criteria into the optimization process. [Kanstren2014a] also provides algorithms for applying optimization to both offline and online model-based test generation using this criteria. Tests are either generated fast at a large scale and a subset best fulfilling the coverage criteria is chosen (offline optimization) or the impact on the coverage criteria is continuously evaluated for the next paths through the test model in parallel to test generation and execution (online optimization). As such optimizations are commonly computationally expensive, while the local test generation and execution environment can be resource limited, techniques for offloading some of this computation to external computation nodes and clusters (e.g., cloud type) have been presented for both online and offline test generation and optimization in [Kanstren2013b].

Beyond optimizing a test set before execution, optimizing it for dynamically evolving targets such as failure location is important. Generated tests have the potential to be very large, and finding the root cause in such large execution and data traces can be an exhaustive task. It is not possible to optimize for such properties beforehand as they are only known *after* the tests have been generated, executed and failures have been observed. Thus, this requires a different type of an optimization approach that uses feedback from the test generation and execution back to the test generator. [Kanstren2014b] presents such an algorithm for minimizing the test trace length required to reach the failure, as well as extracting patterns from the results to describe the conditions under which the failure can occur.

4. Automated testing of highly configurable systems

Many modern systems are highly configurable, meaning that several features of the system can be configured depending on the customers' needs. For example, in software applications running on mobile phones, features can be represented by the type of phone, operating system, installed applications, etc. Each configuration represents a different product. In industrial systems, there can be millions of possible configurations. The availability of composed services in a distributed system, including the types of services, may also change during operation leading to constantly varying configurations. Software failures might appear only with a specific combination of features present in the product. Due to the large number of possible combinations, it is infeasible to manually test all of them.

The two main challenges are: (1) how to represent the variability in an expressive and easy to use way and (2) how to use such variability description to automate the generation of test cases that are effective in revealing failures. Variability can be expressed with several formalisms, as for example models in UML notation or domain specific languages.

4.1. Software Product Lines

There are various ways in which software can be highly configurable. One aspect of configurability is the construction of families of software systems. The best-known approach for this is software product lines (SPL), which can lead to “drastically increasing the productivity of IT-related industries” [Sugumaran2006]. SPL have received particular attention from the research community, with dedicated special issues in Communications of the ACM [Sugumaran2006] and in IEEE Software [McGregor2010]. How to model variability in SPL has been recently surveyed in [Sinnema2007], in which six types of modelling techniques are described. An overview of testing methods for SPL is given in [McGregor2001], [Pohl2006]. Regarding the verification of SPL systems, recent surveys have been carried out to assess the current state of the art [Tevanlinna2004], [Lutz2007], [Lamancha2009]. There are two main conclusions that are drawn in these surveys: (1) very little existing research related to testing of SPL compared to other aspects of SPL (e.g., management and modelling), and (2) empirical analyses on actual industrial systems are particularly rare. This means that, not only there are not many results in testing of SPL, but also for most proposed techniques in the literature there is no evidence to claim their applicability in real-world industrial scenarios, such as the case studies provided by the industrial partners of ATAC. If we consider the situation of year 2012, still many publications in important research venues lack empirical studies on actual industrial systems, as for example [Uzuncaova2010], [Perrouin2010], [Cabral2010], [Mccaffrey2010].

Verifying product lines with model-based testing (MBT) has been introduced in [Kamsties2002], [Kamsties2003], [Kamsties2004]. It describes how to represent use cases with UML extensions and consider the variability of software product lines. This method supports derivation of test cases from these models. The MBT approach ScenTED-method [Reuys2005] provides the derivation of application-specific test scenarios and test cases from use cases and activity charts on the domain level, which contain the variations of the different products. Variation points and variants of the product line are annotated in the use cases. Variant-specific test cases can be automatically generated from the activity diagrams. The FMT approach [Schürr2010] combines feature models with the classification tree method (a black-box testing approach). Feature models are used to describe commonalities and variability in software product lines. Product lines whose products have different feature sets can be tested effectively by modeling individual features separately and then combining the right models for each product [Jääskeläinen2008]. UI-level variance, on the other hand, can be handled by modeling the UI details separately from the functionality of the products [Katara2006]. Both of these techniques require a highly modular modeling approach and support for managing the resulting model components [Jääskeläinen2011]. Other approaches include [Bertolino2003], [Kolb2003], [Kolb2006], [McGregor2001], [Nebut2002], [Olimpiew2005], [Uzuncaova2010]. Scheidemann [Scheidemann2006] describes techniques to select representative configurations in product line testing. An approach that has shown some promise in connection with testing complex and highly-configurable systems is the use of domain-specific languages to construct test models [Kloos2009].

4.2. Service Oriented Architectures

Complex and highly configurable systems are composed of various components and services, and are highly dynamic and distributed from their architectural composition viewpoint. Distribution in practice can be of different types, such as physical distribution over a network or logical distribution inside an embedded system into cohesive and decoupled components and services. A common approach in architecting such systems is the use of service oriented architectures. These systems can be composed of different components and the final configuration is often only known during runtime, it is not static and keeps evolving over time. One approach to address this issue is the generation of models from the runtime system based on information captured using dynamic analysis techniques [Kanstren2010]. For example, previously behavioural models have been (semi-)automatically generated based on observing the runtime behaviour of the implementation. This model has been used as a starting point with the help of a MBT tool to refine an initial generated model, to encode the existing assumptions and understanding of the system behaviour and test them for correctness against the implementation. Bertolini et al. describe approaches to test service-oriented software including [Bartolini2008], [Bartolini2009], [Bertolino2008], [Bertolino2008a-c] an approach to test web services from a WSDL description. An application of their approach is described in [Pascale2009]. Other approaches include [Brenner2007], [Brenner2007a], [Canfora2006], [Greiler2009], [Heckerl2005], [Kaschner2009], [Looker2004], [Offutt2004], and [Papazoglou2007].

4.3. Parametrized Models

Some systems under test may be based on a small number of known components, but these components can be combined into a working system in a variety of ways. For example, a system may consist of variable number of buses, each of which can be connected to a variable number of actuators. Even if tests for an individual component are simple to create, having to create them separately for each different configuration to be tested is hardly sensible.

Test generation may be automated with the use of model-based testing. In fact, with compositional test models it is possible to model different types of components separately and combine the results into a test model for the entire system, thus mimicking the actual structure of the SUT. However, the essential problem remains: a model for a bus connected to three actuators may not work with four, an actuator model hard-coded to connect to bus number one cannot be reused with other buses, and a generic actuator model has to be connected to the correct bus somehow.

Parametrized models provide a practical solution. The model components are created to correspond to specific types of system components and given formal parameters to represent possible variation, such as the connecting bus for an actuator. When the test model is composed, each formal parameter is given a number of values according to the desired configuration. Actual model components are instantiated from the generic ones according to the given values, and the formal parameters within each instance are replaced with the corresponding values. With the formal parameters correctly placed into the parts of the generic model components that define its communication with other components, model composition will connect the right instantiated model components together in the test model. Thus, different configurations can be tested with minimal increase in modelling effort.

4.4. Combinatorial Testing

One field of research in dealing with configurable software is combinatorial testing. The challenge is to find an optimal set of configurations that satisfies maximal coverage criteria for each test case [Grindal05],[Cohen2006]. For example, in pair-wise testing, the goal is to generate a test suite for which each pair of feature values is present at least once in a test case. A generalization of this criterion is t-wise testing. The number of test cases needed to satisfy those criteria is significantly lower than the number of all possible test case combinations. The motivation behind these criteria is that, often, failures are revealed only if just some small combinations of feature values are present in a test case regardless of the value of the other features. Unfortunately, generating minimal test suites satisfying those criteria is a difficult task.

The approach REDUCE combines model-based and combinatorial approaches [Bauer2009]. It aims at reducing the complexity of the test problem represented by the large number of configurations and possible test cases. Combinatorics is used to restrict the number of system configurations and to select a small and valid set of test configurations. Model-based techniques are applied to build a test model that describes the relevant stimulation sequences of the test object. A case study showing potential efficiency improvements using combinatorial test design approaches is described in [Cohen1996]. [Kuhn2006] provides empirical information on the interaction of faults, giving weight to test design approaches that focus on small numbers of interactions, while [Kuhn2008] discusses advanced combinatorial design methods useful for testing. [Perrouin2010] presents a technique to scale constraint solvers to generate t-wise test suites. [Mccaffrey2010] makes comparisons of techniques for pair-wise test suite generation.

5. Search-Based Testing

Apart from combinatorial testing, other approaches that may be useful in dealing with large configuration spaces need to be considered as well. In particular, configuration parameters may in themselves come from large or complicated spaces that are too large to test exhaustively. This problem is also important in normal testing when choosing test data. One of the major challenges associated with choosing test data is that of finding test cases that are effective at finding flaws without requiring an excessive number of tests to be carried out. Formal analytical and classical test generation methods often fail because of the combinatorial explosion of possible interleaving in the execution or functional specification of several properties. Search techniques (as for example genetic algorithms) are designed to find good approximations to the optimal solution in large complex search spaces [DeMillo1991], [McMinn2004]. Moreover, these general-purpose search techniques make very few assumptions about the underlying problem they are attempting to solve. As a consequence, they are useful during the automated generation of effective test cases because they avoid one of the most difficult obstacles with which the software tester is confronted: the need to know in advance what to do for every situation which may confront a program.

For example, in service-oriented architectures, these have been used to test for different input and configuration combinations to produce combinations of inputs, service bindings, and network configurations and server load [DiPenta2007]. This addresses non-functional topics such as response time, throughput and reliability, using goals (fitness functions) such as driving the tests towards producing quality of service violations. However, the application of search algorithms in MBT is still limited in industrial contexts. They have been recently applied for system level MBT of industrial embedded systems [Arcuri2010], [Hemmati2010a], [Hemmati2010b], but there are practically no results in the literature regarding the testing of industrial SPL and service oriented architectures [Harman2009]. Another line of work concerns the usage of constraint-based techniques to master the exploration of a search space in automatic test data generation. In the context of unit testing, several prototype tools such as Euclide [Gotlieb2009] and PathCrawler [Williams2009] were built to demonstrate the interest of innovative constraint-based exploration. However, several challenges remain to demonstrate that these techniques can be applied at system testing level to address the testing problem of highly-configurable systems.

Search-based software testing (SBST) is based on translating testing tasks into optimization problems, and using meta-heuristic search algorithms to achieve these tasks. There are quite many studies showing that the search-based methods in testing are useful (e.g., superior to random testing), but most of these studies are focused on small to medium sized open-source software. This leads to doubts on scalability and applicability of search-based testing to industrial software. Moreover, we are interested in applying these methods on embedded software, which raises further applicability concerns. We hypothesize that SBST can be useful for complex software for embedded systems in the industry, as well. Furthermore, SBST can be enhanced with other methods to form hybrid methods, in order to overcome performance or other limitations.

In their seminal paper, Miller and Spooner published an approach for generating test data in 1976 [Miller1976], which has become known as search-based software testing. Their approach is based on a straight-line variant of a given program that represents the desired execution path. Later in 1990, Korel [Korel1990] extended the idea and introduced the branch distance, which removes the need for the straight-line variant and the problems associated with it.

Search-based software testing efforts are primarily focused on test data generation for the following purposes:

- Structural testing: coverage of specific program structures, e.g., branch coverage
- Functional testing: to find inputs that the software fails to achieve required functionality
- Non-functional testing: for example worst-case execution time
- Grey-box testing: disprove certain intermediate properties of the program, e.g., by exercising assertions in the code

5.1. Structural testing

Most focus in search-based software testing has been in structural testing. Significant work has been published that led to fitness functions for path coverage, branch coverage, dataflow coverage and other types of structural coverage. A common fitness function for branch coverage is the normalized sum of approach level and branch distance, devised by Wegener et al. [Wegener2001]. Consider three nested if statements, where the search is trying to reach the True branch of the innermost conditional, as seen in Figure 5.1.1. Approach level shows how many more conditionals need to be satisfied to reach the target. Assume the variable count is zero. Then the first conditional would evaluate to False, diverting from the target. There are two more conditionals that did not evaluate, so the approach level is 2. If the variable count is bigger than 10, then first conditional would be satisfied but not the second, giving an approach level of 1.

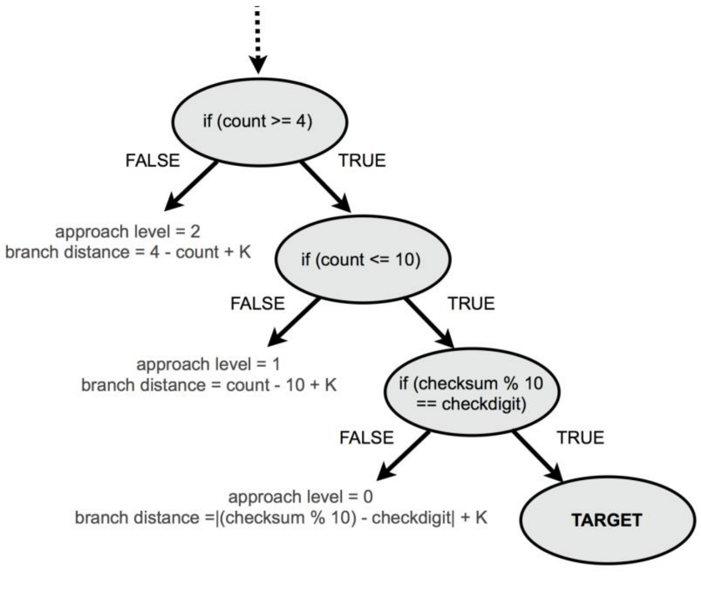
if (count >= 4)
FALSE
TRUE
approach level = 2
branch distance = 4 - count + K
if (count <= 10)
FALSE
TRUE
approach level = 1
branch distance = count - 10 + K
if (checksum % 10 == checkdigit)
FALSE
TRUE
approach level = 0
branch distance =|(checksum % 10) - checkdigit| + K
TARGET

Figure 5.1.1: Approach level and branch distance for three nested if statements.
The example is taken from [McMinn2011].

The branch distance is calculated when execution diverts from the target, at any approach level. It is a measure of how wrong were the variable values to satisfy the conditional. If the first conditional ($\mathtt{count} < 4$) is not satisfied, then the branch distance would be $4 - \mathtt{count} + \mathrm{K}$. Branch distance formula depends on the type of the relational predicate. Tracey et al. [Tracey1998] provides the full list of corresponding formula for different relational predicates (Table 5.1.2).

Relational predicate	Objective function
Boolean	if `True` then 0 else K
$a = b$	if $\|a-b\| = 0$ then 0 else $\|a-b\| + K$
$a \neq b$	if $\|a-b\| \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$a \lor b$	`min(cost(a), cost(b))`
$a \land b$	`cost(a) + cost(b)`
$\neg a$	Negation is moved inwards and propagated over a

Table 5.1.2: Branch distance formula for different types of relational predicates, as devised by Tracey et al. [Tracey1998].

Before approach level and branch distance are summed to form the fitness function, the branch distance needs to be normalized to the range of 0 to 1. As the maximum branch distance is usually not known, a number of different normalization formulas have been suggested. A common one is $1 - \alpha^{-x}$, $\alpha = 1.001$. Arcuri [Arcuri2010] discusses how different normalization formula may affect the search.

5.2. Functional testing

The automated parking system of DaimlerChrysler [Buhler2003, Buhler2008] is probably the most known case study in search-based functional testing. The software for the logical part of automated parking is tested in a simulated environment. Initial conditions of the parking scenario, such as size of the parking space and relative position of the car, are the inputs to the program. The fitness function is the shortest distance to any collision point during the parking manoeuvre. The search algorithm tries to minimize this fitness value (distance to collision), to find a possible input that leads to the car colliding to an object, and hence failing at its function. As in this example, fitness functions are dependent on the functionality that is being tested.

5.3. Temporal testing

Temporal testing refers to running a component and measuring its execution time, to find the worst-case or best-case execution times (WCET/BCET). The maximum and minimum limits of the execution time have great importance for some real-time systems (especially if it is safety critical), as the components not only need to function correctly but also in timely fashion. Search-based software testing has been successfully applied to find test cases that yield higher (or lower) execution times [Wegener1997, Puschner1998]. As opposed to the static timing analysis, search-based methods under-estimates the WCET or over-estimates the BCET. Static analysis tends to be very conservative, so it can be beneficial to use both search-based methods and static analysis to find both upper and lower bounds to WCET or BCET [Puschner1998, Mueller1998].

5.4. Assertion and exception testing

Assertion testing is a form of grey-box testing where structural and functional elements are combined. Programmers may insert assertions into the code, that specify conditions that have to be satisfied at that point in the code. Some assertions can be added automatically, such as division by zero. Korel and Al-Yami [Korel1996b] explain how to translate an assertion into statement coverage. Consider the following assertion code:

```
assert(i > 0 and not (i > x and x > 10));
```

As the search is aimed at finding a test case that falsifies it, the assertion is negated and translated into executable code:

```
/* i <= 0 or (i > x and x > 10) */
if (i <= 0)
      ReportViolation();
if (i > x)
      if (x > 10)
            ReportViolation();
```

After the above translation, the problem of exercising the assertion is reduced to executing one of the ReportViolation() statements, i.e., statement coverage.

Tracey et al. [Tracey2000] applies the same ideas to raise exceptions that are handled in the code. They generate test data for both raising the exception and structural coverage of the exception handling code. Similar to the assertions mentioned above, these problems also get reduced to statement coverage.

5.5. Challenges in Search-Based Software Testing

Testability transformation

In certain cases, the fitness function may not serve as enough guidance for the search. The classical example is the flag problem. When a condition is translated into a boolean variable, and later this boolean is used in the conditional statement, instead of the original predicate, e.g.,

```
bool flag = a > b;
if (flag) { ...
```

In this case, the branch distance will be either 0 or 1, which is not useful for guiding the search. Harman et al. [Harman2002] discusses testability transformation for these cases. Such transformations create a variant of the original problem that is easier to tackle. In the above example, if(flag) is replaced with if(a > b), which leads to a more useful branch distance for the search. The input vector that covers the target branch of the variant, would also reach the same branch in the original program. However, the variant do not need to be functionally equivalent to the original (although it was in this simple example).

Another use of testability transformation is when nested conditionals lead to many local optima, as in the following code:

```
if (a == b) {
c = b + 1;
if (c == 0)
... // target branch
}
```

The search algorithm, guided by the sum of approach level and normalized branch distance as a fitness function, first tries to satisfy the initial condition of a == b. Due to the second branch, the target can be reached only if b is equal to -1. At this point the search would mutate b, but that is very likely to break the initial condition of a == b. So each value pair of (a,b) that satisfies this equality would be a local optima. In the presence of too many local optima, the search becomes inefficient, in this case as bad as random search. McMinn et al. [McMinn2005] apply a testability transformation that combines the two branch distances, as follows:

```
double dist = 0; // extra variable
dist += branch_distance(a, b, "==");
c = b + 1;
dist += branch_distance(c, 0, "==");
if (dist == 0.0)
// target branch
```

The transformed version, by accumulating all the relevant branch distances, does not suffer from the extremely high number of local minima. On the contrary, it leads to a very smooth fitness landscape. An important drawback is that the transformed version is not equivalent to the original code. If the second in the original has some code that should not run when the first conditional is false (e.g., code checking for inequality to zero, to avoid division by zero), this may lead to run time errors. Apart from such problems, McMinn et al. also demonstrates that the nesting testability transformation is speculative, and sometimes may not improve the search efforts.

State-based programs

Most of the work in search-based software testing is focused on procedural code with clear input to output relation, in the form of stateless functions. However, functions, class objects or other components of the program may store data, and behave differently based on this internal state. For example, a particular branch in the code can be infeasible with a global variable's current value. Then the branch coverage problem extends to getting the required value to the global variable, in other words putting the program into the correct state. This becomes an additional challenge, as the branch distance does not give guidance for finding the parts of code that needs to run for setting the global variable with a suitable value.

Baresel et al. [Baresel2003] uses a chaining approach for a single function, where a test case consists of calling the function N many times, rather than once. The fitness value for such sequence is basically the minimum of fitness of each function call. Tonella [Tonella2004] devises a similar approach, but for classes in object-oriented programs.

Another technique used to tackle this problem is the Chaining Approach, which was developed by Ferguson and Korel [Ferguson1996]. The chaining approach is based on analysing the source and finding the nodes in the control flow graph with internal variables that may need to run to put the program in the correct state. McMinn and Holcombe use chaining approach as a fail over mechanism when the normal evolutionary algorithm fails to reach the target branch [McMinn2004].

Execution environment

Often, the program under test interacts with the environment in certain ways. For example, it may read data from a file, the network, or the I/O Bus. The execution may depend on the content of the data read from these sources. Therefore, to be able to divert the program into target branches/paths, relevant elements of the environment need to be simulated or manipulated in some way. In unit testing a common approach is to use mock objects, which mocks the behaviour of an environment structure (e.g., a mockFile object which mocks the original file object of the system library). Automatically generating these mock objects is an important challenge [McMinn2011].

Oracle cost

Even though desirable, in many cases it is very difficult create automated oracles that will tell if the output of running a test case is correct. Instead, human oracles are used, which is costly. Reducing the cost associated with the human oracle is an important research topic. Primarily, this translates to reducing the number of test cases that the human must evaluate while satisfying the test adequacy criteria. In a recent paper, Harman et al. [Harman2010] explain their approach on minimizing the number of test cases while not compromising on the branch coverage.

A second aspect is the length of a single test case. Fraser and Arcuri [Fraser2011] discuss and suggest ways to overcome the problem of test length abnormally growing over time, for testing object oriented software. Lietner et al. [Leitner2007] present a combination of static slicing and delta debugging to efficiently minimize unit test cases.

Proposed solutions

Even though SBST have received reasonably good attention in academia in recent years, it is far from being adopted by the industry as part of the regular testing efforts. In order to gain visibility and acceptance by the industry, experiments of search-based techniques on real-world industrial software should be conducted, so that applicability of SBST in the industrial context can be better understood.

Doganay et al. [Doganay2013] discuss application and performance of search-based techniques on hundreds of unit level test artefacts that are part of industrial embedded software in the railways domain. The particular software under test is written in function block diagrams, a programming language mostly used in embedded control systems. In a related paper, Eduard et al. argue for using both model-based and search-based techniques in parallel for improved structural coverage of function block diagrams, and explain a tooling structure trough a case study [Eduard2013].

In another industrial case study, Doganay et al. explain the difficulties that arise when applying SBST for a complex embedded system in the telecom domain [Doganay2014]. They discuss the special adaptations made to accommodate the specific execution environment. Nested input structures, uninitialized pointers, and other non-trivial input variables are handled by parsing relevant information from existing test cases created by test engineers, which is a novel approach. In general, such problematic input structures remain largely unsupported in the SBST literature [Baars2011].

6. Complex test data generation

Test data generation takes place in various stages of the software development lifecycle and is relevant to all types of testing (black-box, white-box, unit test, integration test, functional test, etc.). This data generation can take different forms, where at the basic level the test and domain experts create test data manually based on analysis of the system under test (SUT). Automated approaches use computers and algorithms to generate test data automatically based on various models of the SUT. These models can take different forms such as using the SUT itself as the model (e.g., source code or interface definitions), using SUT specifications as the model (e.g., UML specifications), creating specific test models (e.g., model programs for a model-based tester), or hybrids of these.

6.1. Basic Approaches

Software is composed of various units and their interrelations. These units are defined by objects such as classes, possible/required relations between classes, and methods defining functions that can be performed by the objects. Data types include the basic primitive types (integers, floats, strings, etc.), and more complex structures such as arrays, lists, classes, and their combinations. Additional properties to consider in test data generation include methods arguments and their combinations, and interactions between software units.

The value generation algorithm depends on the parameter type. For example, if the parameter is of type signed short int (in C++), the tests generator has to create a value from the interval starting with –32,768 and ending with 32,767. The generated value has to be the whole number. Data generation for such primitive types is straightforward – the generated value has to be selected from the given range. For more complex data types different approaches need to be applied. For example, a character string can be any length (although practically limited by the computing environment). Similar approaches as for primitive types can be applied by choosing a string length and generating characters as values from the allowed alphabet. For composed elements such as lists and arrays, the number of elements needs to be similarly defined and a matching generation algorithms applied for each element.

The test data generation for class type structures is more complicated. This entity can be viewed as composed of other simple or complex types, recursively. One approach for this is to use data flattening technique [Meyer1997]. in such approach, one parameter of a complex type is converted into a set of parameters which are of simple types. For example, there is the class Triangle, presented in Figure 6.1 as a part of the 3D renderer software. The class Triangle consists of three attributes: a, b and c. Each attribute represents the triangle edge length and is of a float type. There also is the class Rasterizer with the method Render. The method accepts two input parameters: the triangle and the texture and returns the array of Pixel objects. The triangle class is a complex type.

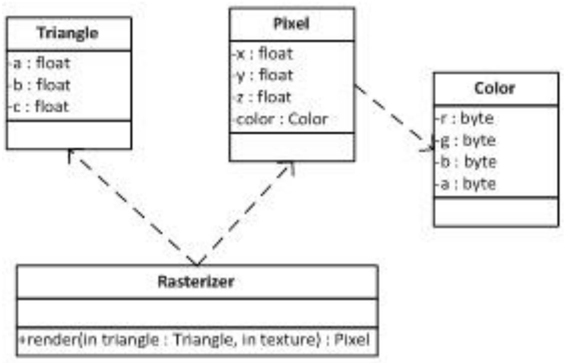

Figure 6.1. The 3d renderer software class diagram

The tests generator cannot generate input values for the parameter triangle. To overcome this, the type flattening is performed and method Render is transformed into the one which accepts 4 parameters: triangle_a, triangle_b, triangle_c and texture.

6.2. White-Box Data Generation

In white-box test data generation we make use of information about the implementation of the SUT, and/or execute the tests directly against the SUT implementation units (vs its external interface). The simplest such test data generation is random test data generation against specific units or modulse. Test data are created by selecting input values randomly [Duran1984] for software under test methods and checking if generator has reached the defined coverage criterion. When this random generation is not guided in any way, the test data contains mostly some meaningless data (from software domain perspective). Various ways to guide the test generation are then used to provide more relevant results.

Feedback analysis based random test generation approaches have been presented by different authors, e.g. [Pacheco2007, Patrice2005]. In these approaches, randomly generated tests are executed and the coverage is calculated after each test execution. Feedback from these executions is used to guide the generation of new values. Choice of new values is randomization based but influenced by how the used values impacted the coverage of the SUT testing as compared to previous tests. The advantage of this approach is that the generation algorithm is quite simple and easy to implement. The drawback of this approach is that the generation is a time consuming process, especially when the software unit under test is quite complex.

Path based tests generation is a white-box test generation strategy aiming to cover different execution paths through the program code. A control-flow graph of the possible execution flows through the SUT is generated, typically using static analysis based tools. Based on approaches in graph theory, test inputs are generated to drive the SUT execution through the different possible paths. To achieve this, techniques such as constraints solving [Gotlieb1998] and relaxation are used [Gupta1998]. These approaches best work when analysing programs with simple data types (float, integer, etc..). More complex data structures and program structures are problematic, such as calling other program units and using variables of complex types (arrays, pointers, data structures, etc..). Some approaches to address these issues have been proposed such as for pointers [Gotlieb2005], calls procedures and/or functions [Sy2003, Korel1996]. For software which uses complex data types, data transformation into equivalent but easier to analyse data types have been proposed [Korel1996, Aleksandar2007]. The advantage of path based tests generation is the possibility to generate the minimal needed tests data set which would satisfy the selected coverage criterion. Also during code analysis the unreachable paths of code could be detected and marked as failures [Beyer2004]. The disadvantage of path based test generation algorithms is that they are quite complex and not always guarantee a full code coverage. In path-based generation, feedback from execution can be also used similar to random testing [Patrice2005].

Beyond using static analysis as input for generation, or combining it with some dynamic execution information, also approaches to white-box test data generation without initial static analysis phase have been developed. The chosen SUT unit is executed with some input data and during its execution runtime parameters are observed: executed paths, executed branches, executed operators. Based on observations, new additional input data are generated in order to drive execution by selected control flow path [Ferguson1996]. Authors are proposing various methods for improving code coverage by tests, such as chaining approach [Ferguson1996], program slicing by diving software unit into separate branches [Hierons1999]. The main drawback of these approaches is that the execution of software has to be performed, which requires the preparation of the whole software infrastructure (environment) – that could not be performed automatically. The advantage is the fast tests generation.

6.3. Model-Based Data Generation

Besides using the SUT implementation as a basis for test data generation, a common approach is to use different types of models of the SUT for test generation. A benefit of such approaches can also be that the implementation is not required for generating the test data as the data can be modelled separately. These models can be expressed at different levels of formalism. Model based tests generation is increasingly becoming more and more important due to the emergence of model driven engineering [Uhl2003] and model driven development [Mellor2003] approaches based software development methods.

A potential source of information for such models is the SUT requirements specification. While these models are commonly created manually, some approaches have been presented to generate them as transformations from requirements specification, even from the textual ones [Gargantini1999]. Data generation in such cases can make use of traditional test data generation techniques such as boundary values analysis and average values analysis.

Formal specifications, such as those expressed in Z notation [Spivey2008] and others [Packevičius2006], strictly define the software functionality. Based on the software formal specification it is possible to generate tests for that software. The formal specifications allow generating not only test data but can also provide an oracle which would be able to determine if software works correctly with given test data. Due to the fact that formal specifications are used for defining critical systems and real time systems, their testing can be alleviated by generated tests from formal specifications [Xin2005]. The disadvantage of such tests creation is that creating formal specifications is expensive and only a few projects are developed using such strategy.

The Unified Modelling Language (UML) [Fowler2003] is semi-formal modelling language. These informal models have some features which could be useful during tests generation. These models are called tests-ready models [Olimpiew2005]. They are usually extended to some extent in order to be suitable for tests generation, For example, UML has testing profile [Baker2007], or an Object Constraint Language (OCL) [Clark2002] model besides UML models could be used for tests generation. Informal models are actively used for testing software developed using product lines approach [Olimpiew2005].

Tests generated using software models usually try to examine such cases as: missing action, incorrect data manipulation by overrunning buffers, incorrect data manipulation between class boundaries, incorrect code logic, incorrect timing and synchronization, incorrect program code sequence execution. During software based on models testing, it is possible to transform models into graphs, such as state graphs. For example, UML diagrams, such as state or sequence can be used for tests generation by transforming them into graphs. For created graphs the usual test generation techniques can be used, the same techniques as for testing software when its code is available [Paradkar2005]. Authors are also proposing to transform models from one language into other ones. Target languages are more suitable for tests generation, for example, the UML models are transformed into SAL models and SAL models are used to generate tests [Kim2005].

Some authors have proposed Jartege tool and a method for random generation of unit tests for Java classes defined in JML (Java Modelling Language) [Oriat2005]. JML allows writing invariants for Java classes and pre and post-conditions for operations. JML specifications are used as a test oracle and for the elimination of irrelevant test cases. Test cases are generated randomly. Proposed method constructs test data using constructors and methods calls for setting state.

6.4. Hybrid Data Generation

The mix of code based and model based tests generation. It is not always possible to have the full specification of software under test. In order to test this software the mix of code based tests generation and model based test generation can be used. For example, such as path finding tool is proposed in [Visser2004]. Data structures are generated from a description of method preconditions. Generalized symbolic execution is applied to the code of the precondition. Test input is found by solving the constraints in the path condition. This method gives full coverage of the input structures in the preconditions. When the code of a system under test is available it is executed symbolically. A number of paths are extracted from the method. An input structure and a path condition define a set of constraints that the input values should satisfy in order to execute the path. Infeasible structures are eliminated during input generation. That is, we can define test models using similar programming language constructs as regular computer programs (such as the SUT), and use the same techniques to analyse and generate data for the models as for the SUT code in a white-box version (e.g, [Veanes2008]).

Another such hybrid approach is discussed in [Beyer2004]. Test data is generated based on code based generation techniques, software is executed with generated test data and it is checked if software has entered the undefined state in the model, or has exceeded restrictions for its variables values [Hessel2006]. Based on code and specification it is possible to verify if code paths executed during testing are defined in the model and allow to check if software has not changed its state to the undefined in the model or has performed illegal transition from one state to another one, thus violating specification [Beyer2004].

7. Automated testing of distributed systems

For distributed system testing there are several methods. Some of the methods are dedicated for almost any distributed systems while others are only dedicated for service oriented architecture and web services or other particular cases. In case of non distributed systems there are a big number of researches taking advantage of SUT (system under test) model checking. Model checking task is to cover all paths through an application. This is also valid for concurrent (multithreaded) applications as displayed by Figure 7.1. This technique is very useful in software testing. It allows detecting and explaining defects, collecting "deep" runtime information like coverage metrics, deducing interesting test vectors and creating corresponding test drivers and many more.

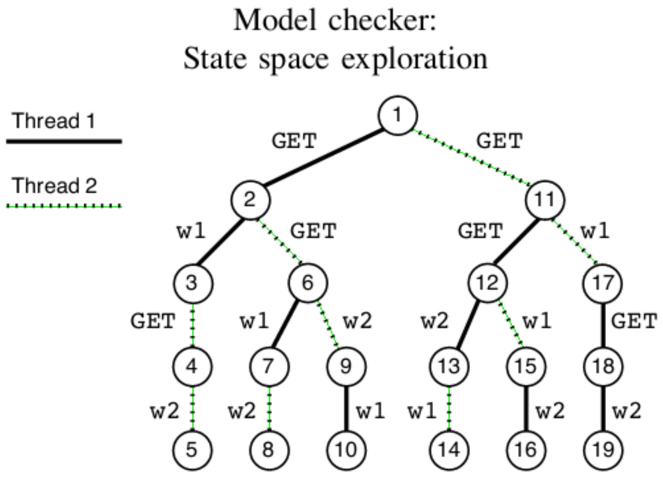

Figure 7.1. State space in the model checker [Artho2009]

In case of distributed systems testing model checking advantage cannot be taken straight forward because the parts of the system are not under control of the same model checker so backtracking becomes unavailable. In this context the term "backtracking" denotes the restoration of a previous state. There are researches covering this topic. One of them proposes using cache-based model checking [Artho2009]. As the name of the article spoils authors suggests using cache while communicating with distributed system parts like distributed servers. Authors of the article call them peers. Authors let SUT be an application executing inside the model checker. Execution of the SUT is subject to backtracking.
As authors states the effects of input/output (I/O) operations with some peers cannot be reversed by backtracking because of different scope. So peers cannot be backtracked because of the two following problems:

1) SUT will re-send data to peer after backtracking. So the peers will become interfered.
2) After backtracking, the SUT will expect external input again from peer but the peer does not re-send previously transmitted data.

As a solution authors suggests execute a single process inside the model checker and run all peers externally. The approach uses I/O cache to relay data between the model checker and it's environment as shown by Figure 7.2. All the actions between I/O operations with peers are treated as atomic actions. All the external I/O operations between SUT and peers are stored to the cache. After backtracking to an earlier program state, data previously received by the SUT is replayed by the cache when requested again. Data previously sent by the SUT is not sent again over the network.

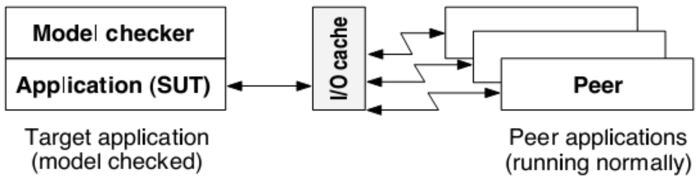
Model checker
Application (SUT)
I/O cache
Peer
Target application (model checked)
Peer applications (running normally)

Figure 7.2. Cache layer architecture [Artho2009]

The cached-based model checking of networked applications approach has three main problems (drawbacks):

1) It can only be used for the systems where every unique request sent to any peer gives the same response. In other words requests do not have effect on each other. This is not true in many distributed systems so none of such systems could be tested using proposed method. Let us take registration to an event system as an example. Clients (users who want to register to the event) send persons data (Name/Surname) to system server (peer). Server responds with SUCCESS/FAILURE message. SUCCESS if registration succeeded, FAILURE in other case (allowed number or users have been exceeded etc). So each request sent to the server changes internal state of the server. It could be database, local variable or anything else. So in this situation proposed method could not be used as each new (excluding first one) registration request would not change the internal state of the server as it should.
2) Data from the cache is replayed by the same process as model checker one while the peers runs on different processes (or even machines). So the I/O operation time may be totally different and parallelism (distribution) factor is destroyed.
3) Proposed method cannot detect defects related to client racing for resources issues. Issues related to resource competition tent to happen when few client instances communicate with the same peers in a distributed system. Proposed method only works/tests one client working with its peers and has no ability to imitate more clients competing for the same resources (peers).

Another model checking based methods uses specification-based verification and validation approach [36]. In this paper authors addresses 3 main web services (WS) testing challenges:

1) When testing third party WS the source code is not available for the tester willing to use the WS. Often only the WS developer has the access to the source code, while the other parties are only interested in the quality of the WS.
2) WS runtime is unknown for the tester. This is one of the major challenges. “This issue is especially serious in WS orchestration, which involves multiple organizations rather than one” states [Tsai2005a]. In this case the number of clients accessing the WS simultaneously is unknown. The way the WS is invoked is unknown as well. “WS testing includes all of the performance, scalability, reliability, availability, security, and stress/load testing aspects for traditional software, but the specialty and distributed property of WS also make WS testing difficult and complicated, and the entire V&V of WS also becomes critical for practical applications”.
3) Testing scope is large because WS consumer may need to choose a WS from hundreds of candidate WS available.

Authors of the article suggest the following development of WS procedure shown by Figure 7.3.

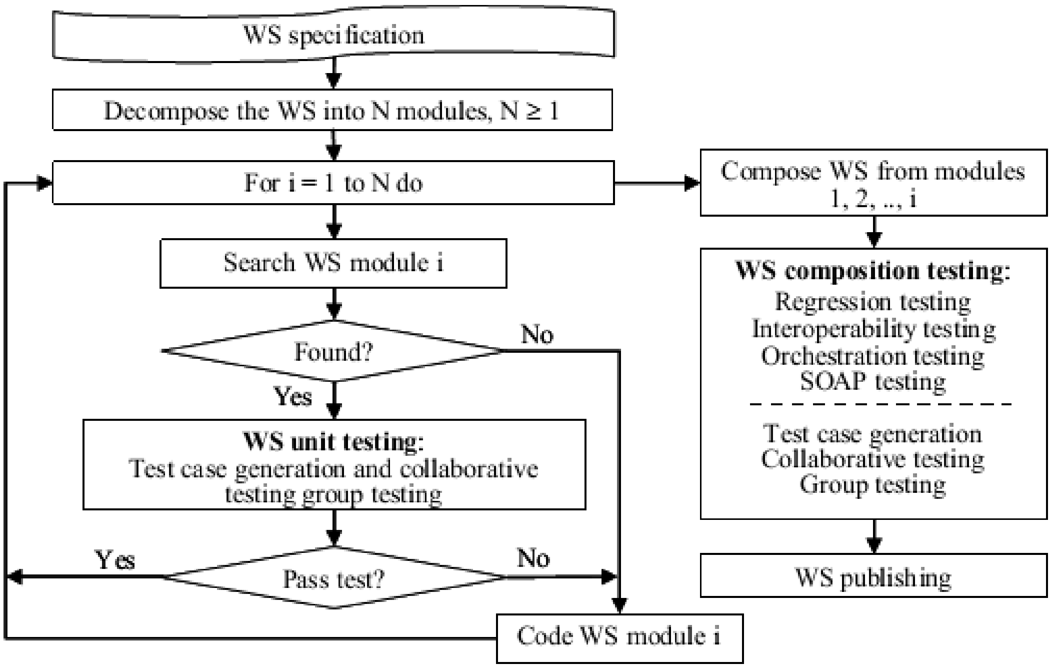

Figure 7.3. Development process of Web Service [Tsai2005a]

Having this, authors suggest specification-based test case generation method. Authors assume that the given WS specification is written in OWL-S. The specifications written in other specification languages should be translated in OWL-S first. First authors suggest doing the verification and validation of the given specification. Authors provided 3 methods for the V&V: Completeness and Consistency (C&C) analysis, model-checking technique based on BLAST [Beyer2004] and verification patterns [Tsai2004].
When the specification passes the test the next step is to use Boolean expression analysis method to extract the full scenario coverage of Boolean expressions [Tsai2003], which are then applied as the input the Swiss Cheese Automated Test Case Generation Tool [Tsai2005b], which, in turn, generates both positive and negative test cases. Positive test cases are used to test if the WS output meets the specification for the legitimate inputs, while negative test cases are used to test the robustness, i.e., the behaviour of the WS if unexpected inputs are applied. Then the test cases are stored in the test case database. Then the following technologies are used in WS unit testing: C&C, model checking and test case generation. Described WS unit testing procedure is shown by Figure 7.4.

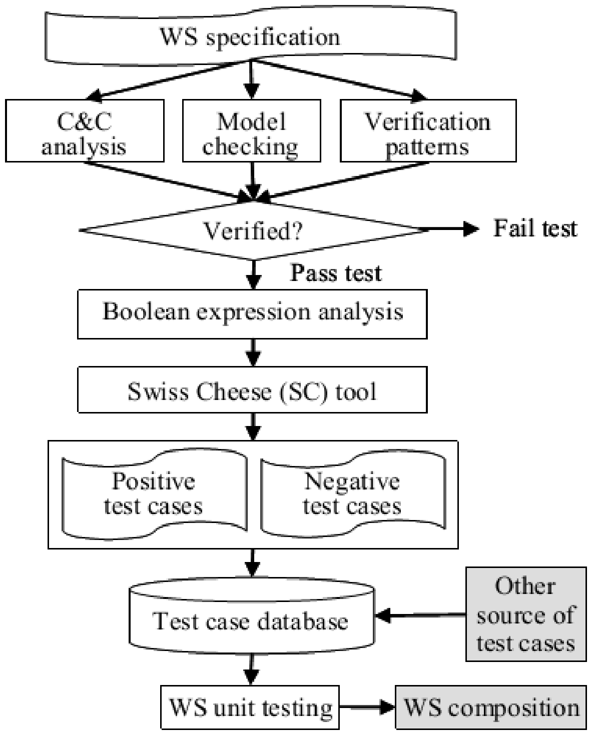

Figure 7.4. WS unit testing [Tsai2005a]

Although the proposed method used few methodologies combining them together the method still has some drawbacks:

1) Completeness and consistency analysis checks the specification only. In many cases the software has the bugs/issues or even inconsistencies with the specification so this technique doesn't actually test the WS but only the specification. So the correctness of the WS itself is not actually tested. This is only good in case one need to filter out the WSs that doesn't meet the given requirements when choosing third party WS to use.
2) In model checking technique case the source code is not used again so the model checker check OWL-S specification rather than the source code. Model checking procedure relies on the conditional or unconditional output, effect and precondition of each atomic/primitive WS to construct their essential inner control logic, which again may not be consistent with actual WS implementation.
3) Test case generation part doesn't provide mechanisms for input data generation in case when data model is relatively large (data model consists of several or more classes that has relations between each other).
4) Only WSs are tested by this method. The clients and WS communication issues are not tested. Also the issues related to concurrency when few clients communicate with the same WS at the same time are not tested at all by suggested method.

Apart from model checking based researches there are other methods on distributed systems (DS) or service-oriented architectures (SOA) testing. One of them is called "An Efficient Formal Testing Approach for Web Service with TTCN-3" [Xiong2012]. Authors of the paper state that often client application and web service itself are developed using different languages or even runs on totally different systems. So in this case the techniques that are designed for some specific language or system cannot be used for such a system testing. To avoid that, authors proposes formal testing approach that uses TTCN-3. TTCN-3 is an international standard test specification and implementation language. It has been developed by ITU and ETSI (European Tele-communication Standards Institute). TTCN-3 intends to support black box testing for reactive and distributed systems. Typical areas of application for TTCN-3 are protocols, services, APIs, and software modules. TTCN-3 specifies test suites at an abstract level [40]. In proposed testing process both server side and client sides should be involved in testing activities as shown by Figure 7.5. Test case analysis and design that is based on models and/or source code of WSs, and ATS specification that is based on WSDL files and test cases, are conducted at server side. The abstract test suite (ATS) will be published via Internet/Intranet, and then it will be retrieved at client sides. The ATS compiling and implementation by developing Test Adapter (TA) and Encoder/Decoder (CoDec) in a native language are performed at client sides. Finally, the test is executed at client sides [Xiong2012].

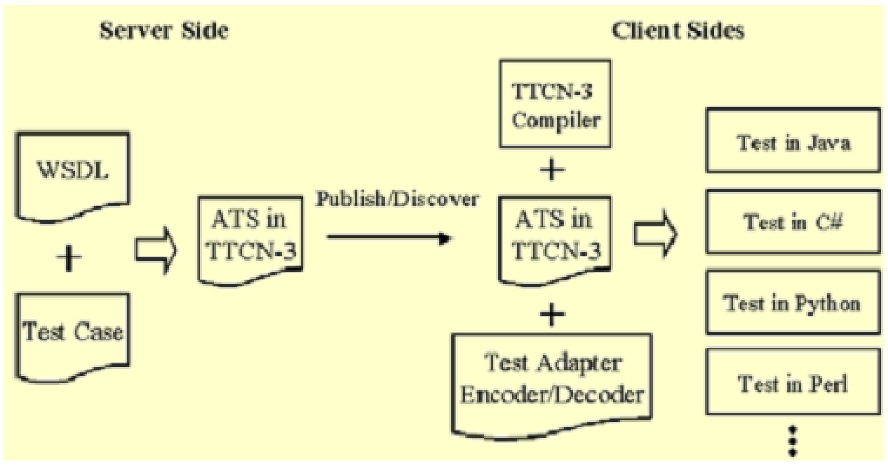

Figure 7.5. Process if Testing Web Services with TTCN-3 [Xiong2012]

Proposed method has the following advantages:

1) "Test case design can be conducted systematically by applying proper methods at server side" [Xiong2012]. This can definitely increase the quality of test case comparing to the ones developed as client sides. Using this method WS actually indicates how it should be tested.
2) Test logic and test implementation are separated, also it is simpler to specify ATS than the platform specific test.
3) Test case itself is done on the server side. So we get maintainability advantage as well as efficiency. We only have to implement the test case once as all clients share the same ATS.
4) The last advantage is related to the third one. Testers at client sides only need to compile ATS, develop test adapter (TA) and Encoder/Decoder (CoDec) in native language. This also works the same for all ATSs.

Although the method has great advantages we can also find few drawbacks of this method:

1) Security is one of the issues. In those cases when ATS is accessible for all the WS users someone with bad intensions can find weak spots of the WS and use them.
2) ATS also describe the test oracle which can already be incorrect. First of all this is a manual task to generate test oracles. Secondly usually it is better to generate test oracles from some specifications or models just for the correctness of the test oracle.
3) The quality of the test cases depends only on the WSs developer. If the test case coverage is low the WS user has no ability to extend or improve this.

4) This technique cannot be applied to existing third party WSs if the WS developer doesn't support this methodology.

None of the above approaches takes advantage of Unified Modeling Language. UML diagrams can be great benefit when testing software. We can find articles that use UML for distributed system testing. One of the articles combines UML and OCL for distributed component based systems testing [Brucker2001]. This method is only suitable for component based distributed systems. Au thors describes such a system as follows: "In our setting with the J2EE/EJB middleware and UML/OCL, we have to consider that the EJB standard requests a split in the interface of the component into two parts: The home interface describing the functions for life–cycle management (such as object creation and destruction) of the EJB and the remote interface describing the functional behaviour. The home interface and remote interface are implemented by the bean implementation. Together, these three parts build an Enterprise Java Bean (EJB), the distributed component in the J2EE model. As we will see later, this interface splitting has a great impact on the organization of the specification and the black–box testing of the EJB." [Brucker2001].

Authors analyse one of the most important UML diagrams: class diagrams. The diagram shows the static structure of the software design: dependencies of classifiers used in the system. In the context of class diagrams, OCL is used for specifying class invariants, preconditio ns and post conditions of class methods. Authors give banking class diagram as an example which is shown in Figure 7.6. Authors presents concept of associations with multiplicities as relations with certain constraints made explicit by appropriate OCL formulae. In the example authors the multiplicities transformed to the following OCL formulae [Brucker2001]:

```
"context Customer
       inv: (1 <= self.accounts.size())
              and (self.accounts.size() <= 99)
context Account
       inv: (1 = self.owner.size())
```

Using invariants for associations, we can also describe if such a relation is partial, injective, surjective or bijec- tive. In our example we would like to express that the associations belongsTo is surjective:

```
context Customer
       inv: self.accounts.forall(a | a.owner = self)
context Account
       inv: self.owner.accounts->includes(self)
```

This guarantees that every account a customer controls (particularly, which is in the set accounts) is owned by this customer." [Brucker2001].

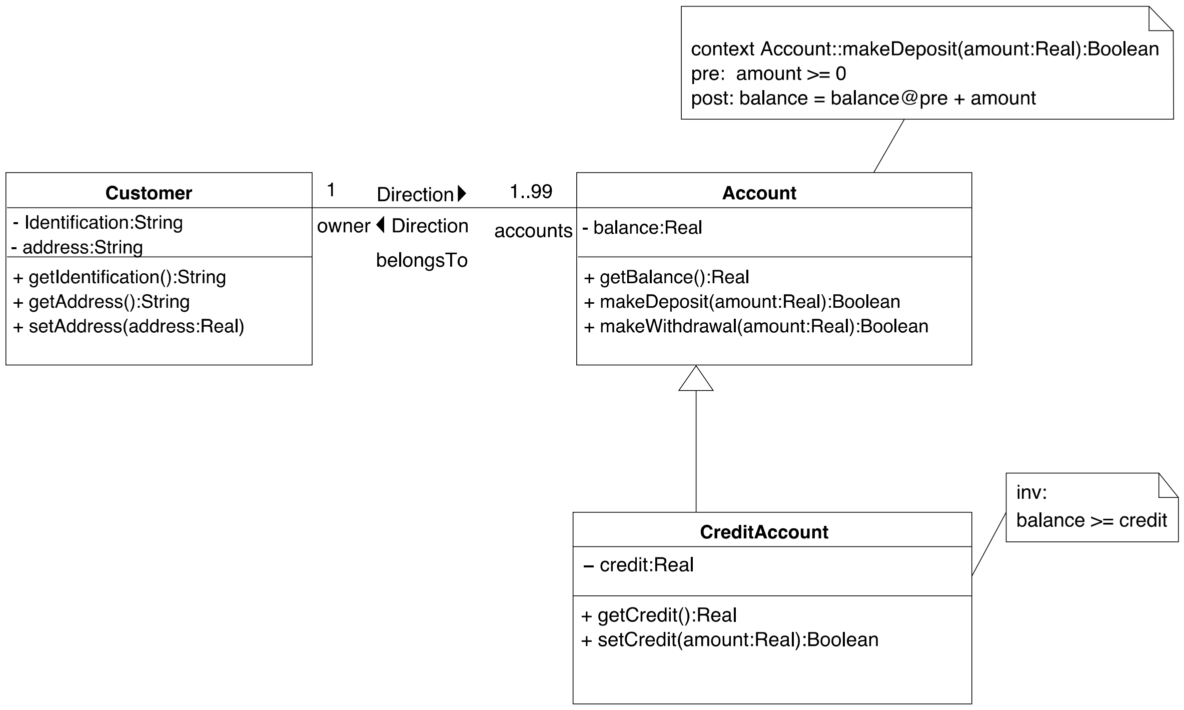
context Account::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
Customer
- Identification:String
- address:String
+ getIdentification():String
+ getAddress():String
+ setAddress(address:Real)
1
Direction
1..99
owner
Direction
belongsTo
accounts
Account
- balance:Real
+ getBalance():Real
+ makeDeposit(amount:Real):Boolean
+ makeWithdrawal(amount:Real):Boolean
inv:
balance >= credit
CreditAccount
− credit:Real
+ getCredit():Real
+ setCredit(amount:Real):Boolean

Figure 7.6. Modeling a simple bank scenario with UML [Brucker2001]

Here we have the collection of OCL constraints so the custom code generation scheme for constraint checking code of individual EJB needs to be developed. Authors suggest a solution for that. Authors suggest using abstract and concrete view of EJB. Also authors choose to use only a very simple data refinement notion. The notion requires that any formula of the abstract view is implied by the formulae of the concrete view. Authors generate code for runtime checking the formulae both on abstract and the concrete view. Having this, authors provide two rules for coding constraint checks:

1) “if only violations against abstract view constraints (but not concrete ones) occur, we can conclude that the abstract view is not a refinement (as it should be)” [Brucker2001].
2) “if only violations against the concrete view constraints occur (but not the abstract ones) the specification of I is too tight for its purpose” [Brucker2001].

An example of abstract and concrete views is given in Figure 7.7.

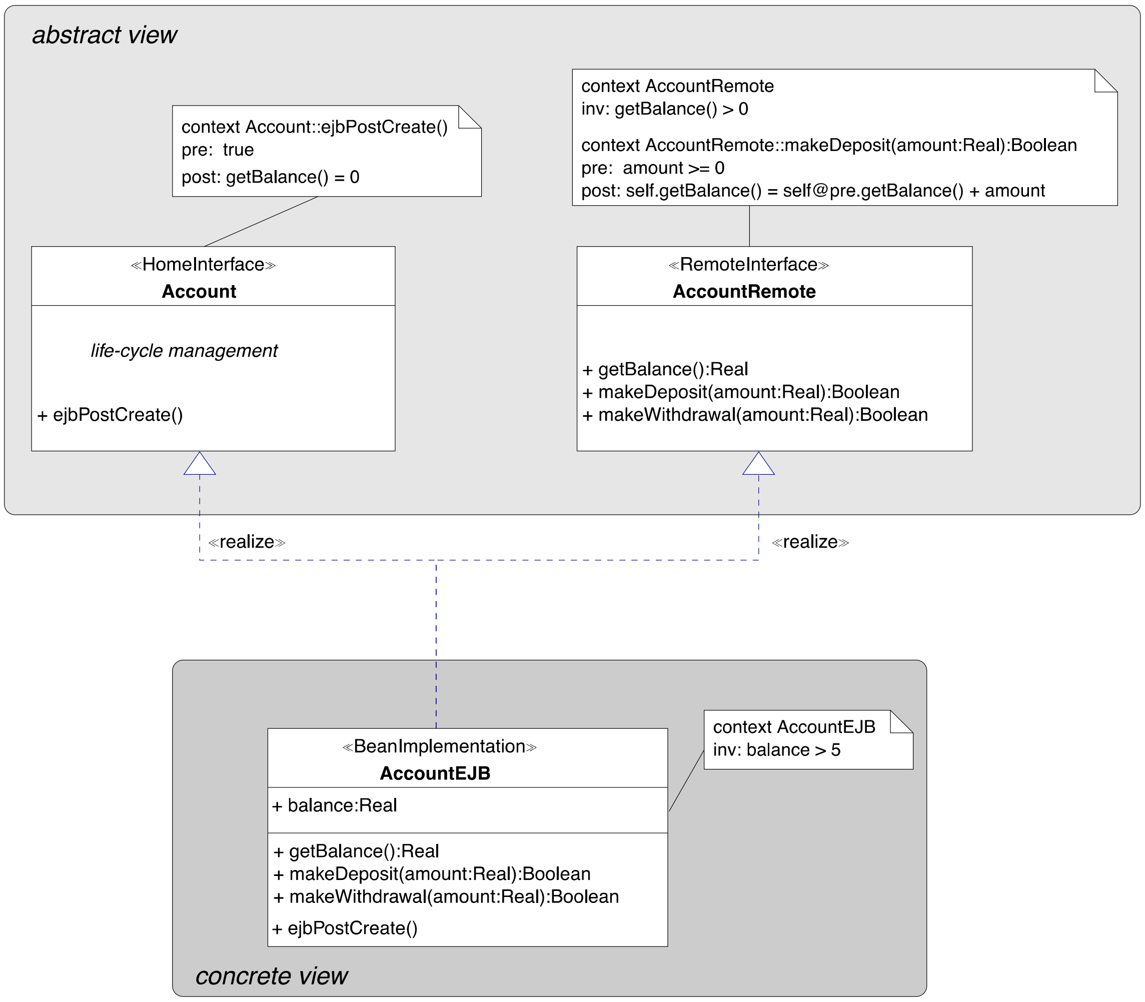

Figure 7.7. Abstract view and concrete view [Brucker2001]

This method has few drawbacks:

1) Only the static UML structure is used in this method. So the dynamic view is not analyzed at all. This is obviously less efficient than checking both structures.
2) The approach is only suitable for J2EE systems.
3) The approach cannot be taken for the existing systems.
4) OCL constraints should be written manually so this testing technique is not fully automatic.

So far we have analyzed Service Oriented Architecture as the architecture with single Web Service. In many real life situations Web Services are combined to create new services by a mechanism called Web Service Composition (WSC). This type of problem is well analyzed by [Endo2008]. “WSC testing is not a trivial task; features like distribution, synchronization and concurrency must be considered during the testing activity” [Endo2008]. Authors of the article propose a test method that applies the Parallel Control Flow Graph (PCFG) model to test Web Services Composition represented in Business Process Execution Language (BPEL). Also authors provide mechanism to analyze test coverage.

Authors at al [Endo2008] analyzes orchestration WSC development paradigm. In this paradigm a central coordinator possesses the control of involved WSs and coordinates the execution of different WS operations, according to pre-established orchestration requirements. The involved WSs should not know that they are part of composition. BPEL is used to define the orchestration of WSs. “In BPEL, the result of a composition is called process, participant WSs are called partners and control structures or commands are called activities” [Endo2008]. BPEL composition can be represented by Figure 7.8.

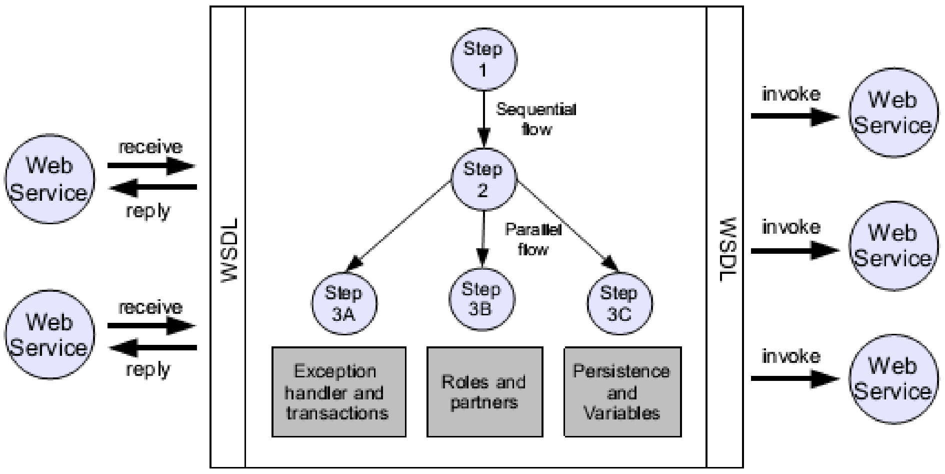

Figure 7.8. BPEL composition [Endo2008]

Graphical WSC example called Loan process presented by Figure 7.9.

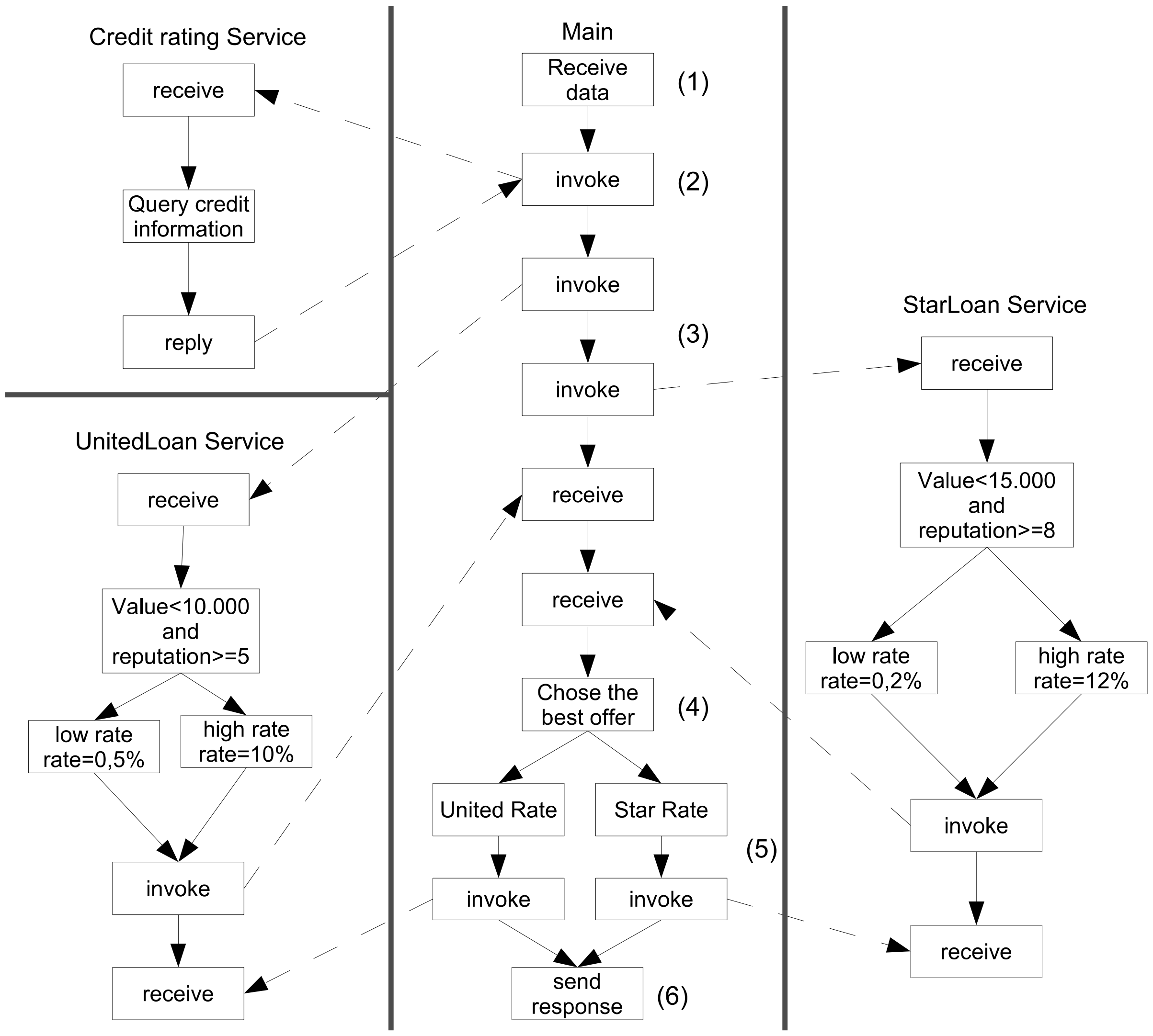

Figure 7.9. Loan Process Example [Endo2008]

When testing using proposed methodology “the first step is to model each BPEL process instance using a traditional CFG. The only difference is that the Receive, Reply, Invoke and Pick activities are modelled as send and/or receive nodes in the PCFG model. For each of these activities, useful information about operation and PartnerLink is recorded and related to the respective created nodes in order to create inter-processes edges in a next step” [Endo2008].

“After this step, inter-processes edges are determined. Using information about PartnerLink and operations of each message passing BPEL activity, we create inter- processes edges between sends and receives of the PCFG model” [Endo2008].

Figure 7.10 represents PCFG of Loan Process Example.

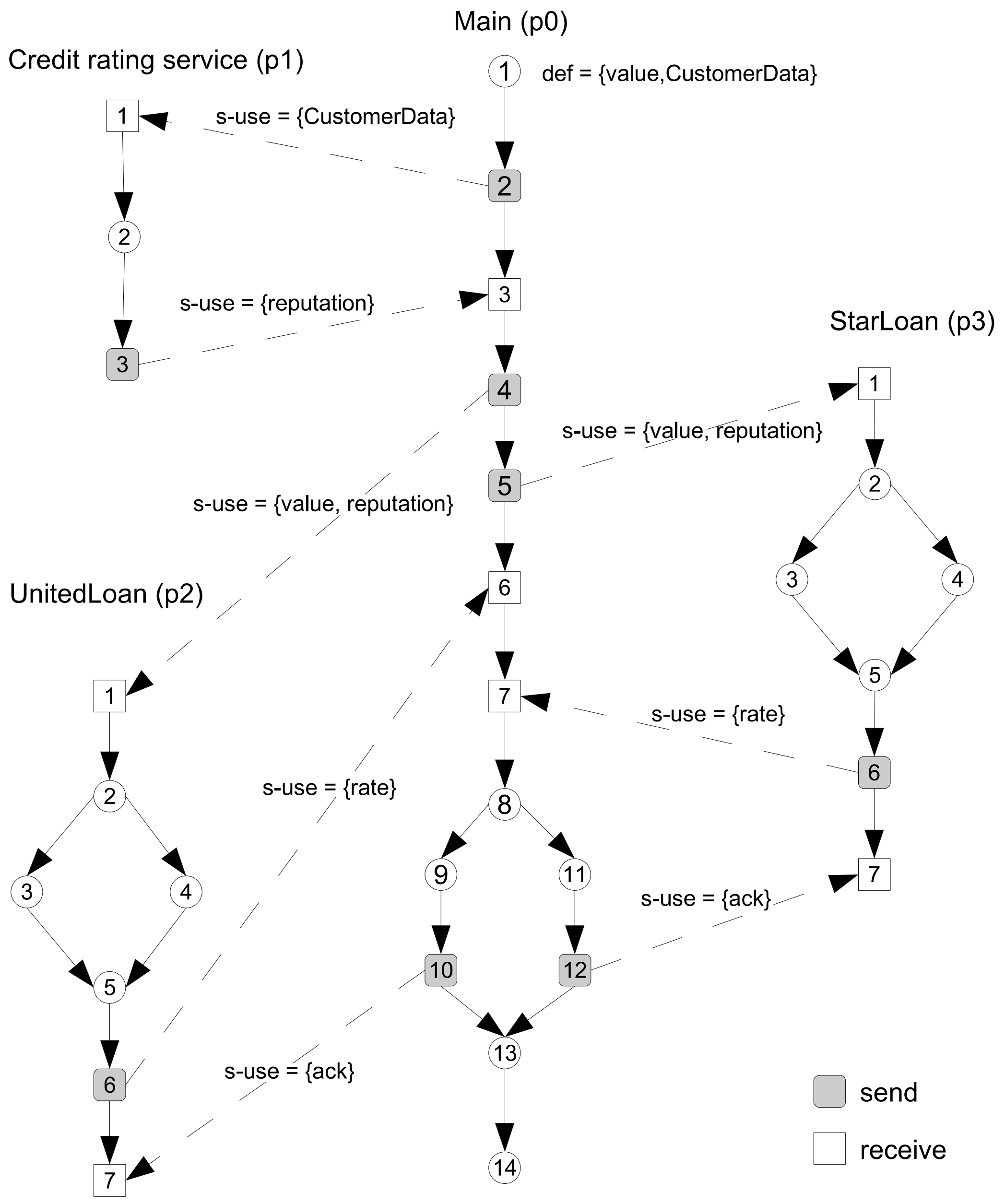

Figure 7.10. PCFG Loan Process Example [Endo2008]

Having PCFG specification we now have all the information about the nodes and edges and so can do the app testing (have all the information about all the paths through the parallel applications). Also the coverage criteria can be provided from this method.

Obviously this methodology has few drawbacks that come out from the information provided earlier:

1) This is only suitable for Web Services Composition. It's not an actual drawback of the method as it is dedicated for this purpose, but in our case it's a drawback.
2) This method doesn't actually solve the Oracle problem. It does provide a mechanism for WSC part execution but for testing additions methods are still required. It can be unit tests of each WS of something similar.

8. References

Ahman2012	D. Ahman and M. Kääramees, "Constrain-Based Heuristic On-line Test Generation from Non-Deterministic I/O EFSMs," in 7th Workshop on Model-Based Testing, 2012.
Aho2011_ITNG	Aho, P. ; Menz, N. ; Räty, T. ; Schieferdecker, I. “Automated Java GUI Modeling for Model-Based Testing Purposes”. 2011 Eighth International Conference on Information Technology: New Generations (ITNG), pp. 268-273, IEEE, 2011.
Aho2011_ICOS	Aho, P. ; Menz, N. ; Räty, T. “Enhancing generated Java GUI models with valid test data”. 2011 IEEE Conference on Open Systems (ICOS), pp. 310-315, IEEE, 2011.
Aho2013_CoDIT	P. Aho, N. Menz, and T. Räty, "Dynamic Reverse Engineering of GUI Models for Testing", Proc. 2013 Int. Conf. on Control, Decision and Information Technologies (CoDIT'13), 6-8 May 2013, Hammamet, Tunisia, pp. 441-447.
Aho2013_EESSMod	Aho, P. ; Suarez, M. ; Kanstrén, T. ; Memon, A. “Industrial Adoption of Automatically Extracted GUI Models for Testing”. International Workshop on Experiences and Empirical Studies in Software Modelling (EESSMod 2013).
Aho2014_ICSTW	P. Aho, M. Suarez, T. Kanstren, and A.M. Memon, “Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing”, Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC PART), 4 Apr 2014, Cleveland, OH, USA.
Aleksandar2007	Aleksandar M, Sasa M, Darko M, Sarfraz K. Korat: A Tool for Generating Structurally Complex Test Inputs. Proceedings of the 29th international conference on Software Engineering: IEEE Computer Society; 2007.
Amalfitano2011	D. Amalfitano, A. R. Fasolino and P. Tramontana, “A GUI Crawling-Based Technique for Android Mobile Application Testing”, 3rd Int. Workshop on Testing Techniques & Experimentation Benchmarks for Event-Driven Software, IEEE CS Press, 2011, pp. 252-261.
Ammann2002	Paul Ammann, Paul E Black, and Wei Ding. Model Checkers in Software Testing. In NIST-IR 6777, National Institute of Standards and Technology Report, 2002.
Ammann2003	Paul Ammann, Jeff Offutt, and Hong Huang. Coverage Criteria for Logical Expressions. In 14th International Symposium on Software Reliability Engineering, pages 99–107. IEEE, 2003.
AmmannOffutt	Paul Ammann and Jeff Offutt. Introduction to Software Testing. In Cambridge University Press.
Arcuri2010	A. Arcuri. It does matter how you normalise the branch distance in search based software testing. pages 205–214. IEEE, April 2010.
Artho2009	C. Artho WL, M. Hagiya, Y. Tanabe, M. Yamamoto. Cache-based Model Checking of Networked Applications: From Linear to Branching Time. ACE 2009.
Baars2011	Baars, A., Lakhotia, K., Vos, T.E.J., Wegener, J.: Search-based testing, the underlying engine of future internet testing. In Proceedings of the 2011 Federated Conference on Computer Science and Information Systems (FedCSIS’11), 2011.
Baker2007	Baker P, Dai ZR, Grabowski J, Haugen O, Schieferdecker I, Williams C. Model-Driven Testing: Using the UML Testing Profile: Springer-Verlag Berlin and Heidelberg GmbH & Co. K; 2007.
Baresel2003	A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. GECCO’03, page 2428–2441, Berlin, Heidelberg, 2003. Springer-Verlag.
Black2000	Paul Black. Modeling and Marshaling: Making Tests from Model Checker Counter-examples. In Proceedings of the 19th Digital Avionics Systems Conference, volume 1, pages 1B3–1. IEEE, 2000.
Beyer2004	Beyer D, Chlipala AJ, Majumdar R. Generating tests from counterexamples. 26th International Conference on Software Engineering (ICSE'04) 2004. p. 326-35.
Blackburn2002	M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars Polar Lander Fault Identification

	Using Model-Based Testing," in Proceedings of the 8th IEEE Internation Conference on Engineering of Complex Computer Systems (ICECCS02), 2002, pp. 163-169.
Bowring2004	Bowring, J. F., Rehg, J. M. and Harrold, M. J. “Active Learning for Automatic Classification of Software Behaviour.” In: Proceedings of the International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, 2004. Pp. 195–205.
Buhler2003	O. Bühler and J. Wegener. Evolutionary functional testing of an automated parking system. 2003.
Buhler2008	O. Bühler and J. Wegener. Evolutionary functional testing. Computers & Operations Research, 35(10):3144–3160, October 2008.
Bringman2008	E. Bringmann and A. Krämer, "Model-Based Testing of Automotive Systems," in Proceedings of the International Conference on Software Testing, Verification and Validation, 2008, pp. 485-493.
Brucker2001	A. D. Brucker BW. Testing Distributed Component Based Systems Using UML/OCL. IEEE 2001:608-14.
Clark2002	Clark T, Warmer J. Object Modeling with the OCL: The Rationale behind the Object Constraint Language (Lecture Notes in Computer Science): Springer; 2002.
Corno2004	Corno F, Sanchez E, Reorda MS, Squillero G. Automatic test program generation: a case study. Design & Test of Computers, IEEE 2004;21:102-9.
Doganay2013	Kivanc Doganay, Markus Bohlin, Ola Sellin. Search Based Testing of Embedded Systems Implemented in IEC 61131-3: An Industrial Case Study. International Conference on Software Testing, Verification and Validation (ICST) - Search-Based Software Testing (SBST), March 2013.
Doganay2014	Kivanc Doganay, Sigrid Eldh, Wasif Afzal, and Markus Bohlin. Search-based Testing for Embedded Telecom Software with Complex Input Structures. The 26th IFIP International Conference on Testing Software and Systems (ICTSS), 2014 (To appear).
Duran1984	Duran JW, Ntafos SC. An evaluation of random testing. IEEE transactions on software engineering 1984;10:438-44.
Eduard2013	Eduard Paul Enoiu, Kivanc Doganay, Markus Bohlin, Daniel Sundmark, Paul Pettersson. MOS: An Integrated Model-based and Search-based Testing Tool for Function Block Diagrams. 35th International Conference on Software Engineering (ICSE) - First International Workshop on Combining Modelling and Search-Based Software Engineering, May 2013.
Endo2008	A. T. Endo AdSS, S. do R. S. de Souza, P. S. L. de Souza. Web Services Composition Testing: a Strategy Based on Structural Testing of Parallel Programs. IEEE DOI 101109/TAIC-PART20089 2008.
Engels1997	Andre Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Tools and Algorithms for the Construction and Analysis of Systems, pages 384-398. Springer, 1997.
Fard2013	Fard, A.M.,and Mesbah, A., “Feedback-Directed Exploration of Web Applications to Derive Test Models”, Proc. International Symposium on Software Reliability Engineering (ISSRE), 2013.
Ferguson1996	Ferguson R, Korel B. The chaining approach for software test data generation. ACM Trans Softw Eng Methodol 1996;5:63-86.
Fowler2003	Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Third Edition ed. Boston: Addison-Wesley Professional; 2003.
Fraser2009	Gordon Fraser, Franz Wotawa, and Paul E Ammann. Testing with Model Checkers: a Survey. In Journal on Software Testing, Verification and Reliability, volume 19, pages 215-261. Wiley Online Library, 2009.
Fraser2011	G. Fraser and A. Arcuri. It is not the length that matters, it is how you control it. In 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), pages 150–159. IEEE, March 2011.
Gargantini1999	Gargantini A, Heitmeyer C. Using model checking to generate tests from requirements specifications. Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. Toulouse, France: Springer-Verlag; 1999. p. 146-62.

Gotlieb1998	Gotlieb A, Botella B, Rueher M. Automatic test data generation using constraint solving techniques. Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis. Clearwater Beach, Florida, United States: ACM Press; 1998. p. 53-62.
Gotlieb2005	Gotlieb A, Denmat T, Botella B. Goal-oriented test data generation for programs with pointer variables. 29th Annual International Computer Software and Applications Conference (COMPSAC'05)2005. p. 449-54 Vol. 2.
Grieskamp2010	Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman, "Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology," Journal of Software Testing, Verification and Reliability, 2010
Grilo2010	A.M.P. Grilo, A.C.R. Paiva, and J.P. Faria, “Reverse Engineering of GUI Models for Testing”, 5th Iberian Conference on Information Systems and Technologies (CISTI), Santiago de Compostela, Spain, 2010.
Gupta1998	Gupta N, Mathur AP, Soffa ML. Automated test data generation using an iterative relaxation method. Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. Lake Buena Vista, Florida, United States: ACM Press; 1998. p. 231-44.
Halpern2002	B. Hailpern and P. Santhanam, “Software debugging, testing, and verification”, IBM SYSTEMS JOURNAL, VOL 41, NO 1, 2002
Haran2007	Haran, M., Karr, A., Last, M., Orso, A., Porter, A. A., Sanil, A. and Fouché, S. “Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks.” IEEE Transactions on Software Engineering, Vol. 33, No. 5, pp. 287–304, May 2007.
Harmanani2005	Harmanani H, Karablieh B. A hybrid distributed test generation method using deterministic and genetic algorithms. Fifth International Workshop on System-on-Chip for Real-Time Applications (IWSOC'05) 2005. p. 317-22.
Hessel2006	Hessel A, Pettersson P. Model-Based Testing of a WAP Gateway: an Industrial Study. FMICS and PDMC2006.
Hierons1999	Hierons R, Harman M, Danicic S. Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability 1999;9:233-62.
Hinchey2012	M. Hinchey and L. Coyle, “Conquering Complexity”, 466 pages, Springer
Hong2002	Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic-Based Theory of Test Coverage and Generation. In Tools and Algorithms for the Construction and Analysis of Systems, pages 327–341. Springer, 2002
Jin2008	Jin, H., Wang, Y., Chen, N.-W., Gou, Z.-J. and Wang, S. “Artificial Neural Network for Automatic Test Oracles Generation.” In: International Conference on Computer Science and Software Engineering, Wuhan, Hubei, 2008.Pp. 727–730.
Jääskeläinen2008	Antti Jääskeläinen, Antti Kervinen, and Mika Katara, “Creating a test model library for GUI testing of smartphone applications,” in Proceedings of the 8th International Conference On Quality Software (QSIC 2008) (short paper), Hong Zhu, Ed. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2008, pp. 276–282.
Jääskeläinen2009	Antti Jääskeläinen, Mika Katara, Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, Tommi Takala, and Heikki Virtanen, “Automatic GUI test generation for smartphone applications – an evaluation,” in Proceedings of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009). Los Alamitos, CA, USA: IEEE Computer Society, May 2009, pp. 112–122 (companion volume).
Jääskeläinen2011	Antti Jääskeläinen, “Design, implementation and use of a test model library for GUI testing of smartphone applications,” Doctoral dissertation, Tampere University of Technology, Tampere, Finland, Jan. 2011, number 948 in publications.
Kalpana2005	Kalpana P, Gunavathi K. A novel specification based test pattern generation using genetic algorithm and wavelets. 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05) 2005. p. 504-7.
Kanstren2014c	T. Kanstrén, P. Aho, “Towards a Holistic Architecture for a SIP Test Framework”, 6th International Conference on Advances in System Testing and Validation Lifecycle (VALID 2014), October 12-16, Nice, France, 2014.
Kanstren2014b	T. Kanstrén, M. Chechik, “Trace Reduction and Pattern Analysis to Assist Debugging in Model-Based Testing”, Submitted to 5th IEEE International Workshop on Program Debugging, November 3, Naples, Italy, 2014.

Kanstren2014a	T. Kanstrén, M. Chechik, “A Comparison of Three Black-Box Optimization Approaches to Model-Based Testing”, 5th International Workshop on Automating Test Case Design, Selection and Evaluation (ATSE’14), September 7, Warsaw, Poland, 2014.
Kanstren2013b	T. Kanstrén, T. Kekkonen, "Distributed Online Test Generation for Model-Based Testing", The 20th Asia-Pacific Software Engineering Conference (APSEC 2013), December 2-5, Bangkok, Thailand, 2013.
Kanstren2013	T. Kanstrén, “A Review of Domain-Specific Modelling and Software Testing”, Proc. 8th International Multi-Conference on Computing in the Global Information Technology, 2013.
Kanstren2012a	T. Kanstrén, O-P. Puolitaival, V-M. Rytky, A. Saarela, J. Keränen, "Experiences in Setting up Domain-Specific Model-Based Testing", In Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT 2012), March, 19-21, 2012.
Kanstren2012b	T. Kanstrén, O-P. Puolitaival, "Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation", In Proceedings of the 7th Workshop on Model-Based Testing (MBT 2012), March, 25, 2012.
Kanstren2011	T. Kanstrén, O-P. Puolitaival, J. Perälä, "An Approach to Modularization in Model-Based Testing", In Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011), October, 23-29, 2011.
Kanstren2010	T. Kanstrén, “A Framework for Observation Based Modelling”, VTT Publications, 2010.
Katara2006	Mika Katara, Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mikko Satama, “Towards deploying model-based testing with a domain-specific modeling approach,” in Proceedings of the 1st Testing: Academic & Industrial Conference – Practice and Research Techniques (TAIC PART 2006). Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2006, pp. 81–89.
Kervinen2005	Antti Kervinen, Mika Maunumaa, Tuula Pääkkönen, and Mika Katara, “Model-based testing through a GUI,” in Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), ser. Lecture Notes in Computer Science, Wolfgang Grieskamp and Carsten Weise, Eds., vol. 3997. Berlin, Heidelberg: Springer, Jul. 2005, pp. 16–31.
Kim2005	Kim SK, Wildman L, Duke R. A UML approach to the generation of test sequences for Java-based concurrent systems. 2005 Australian Software Engineering Conference (ASWEC'05) 2005. p. 100-9.
Korel1990	B. Korel. Automated software test data generation. IEEE Transactions on Software Engineering, 16(8):870–879, August 1990.
Korel1996	Korel B. Automated test data generation for programs with procedures. Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis. San Diego, California, United States: ACM Press; 1996. p. 209-15.
Korel1996b	Korel B, Al-Yami AM. Assertion-oriented automated test data generation. Proceedings of the 18th international conference on Software engineering. Berlin, Germany: IEEE Computer Society; 1996. p. 71-80.
Leitner2007	A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. ASE ’07, page 417–420, New York, NY, USA, 2007. ACM.
Lorenzoli2008	Lorenzoli, D., Mariani, L. and Pezzè, M. “Automatic Generation of Software Behavioral Models.” In: Proceedings of the 30th International Conference on Software Engineering (ICSE08), Leipzig, Germany, 2008. Pp. 501–510.
Mariani2007	Mariani, L., Papagiannakis, S. and Pezzé, M. “Compatibility and Regression Testing of COTS-Component-Based Software.” In: Proc. 29th International Conference on Software Engineering (ICSE’07), 2007.
McMinn2004	P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes In Computer Science, 3103:1363-1374, 2004.
McMinn2005	P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. 2005.
McMinn2011	P. McMinn. Search-based software testing: Past, present and future. In2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 153–163, March 2011.

Mellor2003	Mellor SJ, Clark AN, Futagami T. Model-driven development - Guest editor's introduction. Software, IEEE 2003;20:14-8.
Memon2003	A. M. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: reverse engineering of graphical user interfaces for testing”, Proc. 10th Working Conference on Reverse Engineering (WCRE'03). IEEE Comp Society, Washington DC, USA.
Memon2004	Memon, A. and Xie, Q. “Using Transient/Persisten Errors to Develop Automated Test Oracles for Event-Driven Software.” In: Proceedings of the 19th International Conference on Automated Software Engineering, 2004.
Memon2007	A. M. Memon, “An event-flow model of GUI-based applications for testing”, Software Testing, Verification and Reliability, Volume 17, Issue 3 (Sep 2007).
Mesbah2012	Mesbah, A., Deursen, A. van, and D. Roest, “Invariant-Based Automatic Testing of Modern Web Applications”. IEEE Transactions on Software Engineering, vol. 38, no. 1, 2012.
Meyer1997	Meyer B. Object-oriented software construction: Prentice-Hall, Inc.; 1997.
Miao2010	Y. Miao, and X. Yang, “An FSM based GUI Test Automation Model”, 11th Int. Conf. Control, Automation, Robotics and Vision Singapore, 7-10th Dec 2010.
Mikucionis2004	M. Mikucionis, K. Larsen and B. Nielsen, "T-Uppaal: Online Model-Based Testing of Real-Time Systems," in 19th International Conference on Automated Software Engineering (ASE 2004), 2004.
Miller1976	W. Miller and D. L Spooner. Automatic generation of floating-point test data. IEEE Transactions on Software Engineering, SE-2(3):223– 226, September 1976.
Miller2012	T. Miller, P. Strooper, “A case study in model-based testing of specifications and implementations”, Software Testing, Verification, and Reliability, vol. 22, no. 1, 2012, pp. 33-63.
Mueller1998	F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. pages 144–154. IEEE, June 1998.
Olimpiew2005	Olimpiew EM, Gomaa H. Model-based testing for applications derived from software product lines. Proceedings of the first international workshop on Advances in model-based testing. St. Louis, Missouri: ACM Press; 2005. p. 1-7.
Oriat2005	Oriat C. Jartege: a tool for random generation of unit tests for java classes. First international conference on Quality of Software Architectures and Software Quality. Erfurt, Germany: Springer-Verlag; 2005. p. 242--56.
Pacheco2007	Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-Directed Random Test Generation. Proceedings of the 29th international conference on Software Engineering: IEEE Computer Society; 2007. p. 75-84.
Packevičius2006	Packevičius Š, Kazla A, Pranevičius H. Extension of PLA Specification for Dynamic System Formalization. Information Technology and Control 2006;3:235-42.
Paradkar2005	Paradkar A. Case studies on fault detection effectiveness of model based test generation techniques. Proceedings of the first international workshop on Advances in model-based testing. St. Louis, Missouri: ACM Press; 2005. p. 1-7.
Patrice2005	Patrice G, Nils K, Koushik S. DART: directed automated random testing. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. Chicago, IL, USA: ACM Press; 2005. p. 213-23
Pradel2012	Pradel, Michael, Gross, Thomas R., “Leveraging test generation and specification mining for automated bug detection without false positives”, 2012 34th International Conference on Software Engineering (ICSE)
Pretschner2005	A. Pretschner et al., "One Evaluation of Model-Based Testing and its Automation," in Proceedings of the 27th International Conference on Software Engineering, St. Louis, Missouri, USA, 2005, pp. 392-401.
Puolitaival2011	O-P. Puolitaival, T. Kanstrén, V-M. Rytky, and A. Saarela, "Utilizing Domain-Specific Modelling for Software Testing," in 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID2011), 2011.

Puschner1998	P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. pages 134–143. IEEE, December 1998.
Rayadurgam2001	Sanjai Rayadurgam and Mats PE Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In International Conference and Workshop on the Engineering of Computer Based Systems, pages 83–91. IEEE, 2001
Rayadurgam2003	S Rayadurgam and MPE Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In NASA Goddard Software Engineering Workshop Proceedings, pages 91– 96. IEEE, 2003.
Santos-Neto2008	P. Santos-Neto, R. Resende, and C. Pádua, "An Evaluation of a Model-Based Testing Method for Information Systems," in ACM symposium on Applied computing, 2008, pp. 770-776.
Seesing2006	Seesing A, Gross H-G. A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software. International Transactions on System Science and Applications 2006;1:127-34.
Silva2010	J.C. Silva, C. Silva, R.D. Gonalo, J. Saraiva, and J.C. Campos, “The GUISurfer tool: towards a language independent approach to reverse engineering GUI code”, Proc. 2nd ACM SIGCHI symposium on Engineering interactive computing systems, Berlin, Germany, 2010, pp. 181-186.
Spivey2008	Spivey JM. Understanding Z: A Specification Language and Its Formal Semantics: Cambridge University Press; 2008.
Staats2012	Staats, Matt, Hong, Shin, Kim, Moonzoo, Rothermel, Gregg, “Understanding user understanding: determining correctness of generated program invariants”, Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012
Sy2003	Sy NT, Deville Y. Consistency techniques for interprocedural test data generation. Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. Helsinki, Finland: ACM Press; 2003. p. 108-17.
Tracey1998	N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. pages 285–288. IEEE, October 1998.
Tracey2000	N. Tracey, J. Clark, and J. Mcdermid. Automated test-data generation for exception conditions. Software – Practice and Experience, 30:61-79, 2000.
Tretmans2002	J. Tretmans, E. Brinksma, "Côte de Resyste – Automated Model Based Testing", in Proceedings of Progress 2002 – 3rd Workshop on Embedded Systems, 2002, pp. 246–255.
Tonella2004	P. Tonella. Evolutionary testing of classes. ACM SIGSOFT Software Engineering Notes, 29:119, July 2004.
Tonella2012	Tonella, Paolo, Marchetto, Alessandro, Nguyen, Cu Duy, Jia, Yue, Lakhotia, Kiran, Harman, Mark Finding the Optimal Balance between Over and Under Approximation of Models Inferred from Execution Logs, 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation
Tsai2003	W.T. Tsai LY, Feng Zhu and Ray J. Paul. Rapid Verification of Embedded Systems Using Patterns. COMPSAC 2003:466-71.
Tsai2004	[37] W. T. Tsai RP, L. Yu, X . Wei, and F. Zhu. Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems. Software evolution with UML and XML2004.
Tsai2005a	W.T. Tsai YC, R. Paul. Specification-Based Verification and Validation of Web Services and Service-Oriented Operating Systems. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems 2005.
Tsai2005b	W. T. Tsai XW, Y. Chen, B. Xiao, R. Paul, and H. Huang. Developing and Assuring Trustworthy Web Services. 7th International Symposium on Autonomous Decentralized Systems (ISADS) 2005:91-8.
Uhl2003	Uhl A. Model driven arcitecture is ready for prime time. Software, IEEE 2003;20:70, 2.
Utting2006	M. Utting, and B. Legeard, “Practical Model-Based Testing: A Tool Approach”, Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 2006.
Utting2012	Utting, M., Pretschner, A., Legeard, B., “A Taxonomy of Model-Based Testing Approaches”, Software Testing,

	Verification and Reliability, 2012.
Veanes2005	M. Veanes, C. Campbell, W. Schulte and N. Tillmann, "Online Testing with Model Programs," in ESEC/FSE-13, 2005.
Veanes2008	M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann and L. Nachmanson, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer," Formal Methods of Testing, pp. 39-76, 2008.
Vieira2008	M. Vieira et al., "Applying Model-Based Testing to Healthcare Products: Preliminary Experiences," in Proceedings of the 30th International Conference on Software Engineering (ICSE08), Leipzig, Germany, 2008, pp. 669-671.
Visser2004	Visser W, Pasareanu CS, Khurshid S. Test input generation with java PathFinder. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. Boston, Massachusetts, USA: ACM Press; 2004. p. 97-107.
Wegener1997	J. Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. Testing real-time systems using genetic algorithms. Software Quality Con- trol, 6(2):127–135, October 1997.
Wegener2001	J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. Information and Software Technology, 43(14):841–854, December 2001.
Xie2005	Q. Xie, and A. M. Memon, “Rapid crash testing for continuously evolving GUI-based software applications”, Proc. 21st IEEE Int. Conf. on Software Maintenance (ICSM'05), IEEE Computer Society, Washington DC, USA, 473-482.
Xin2005	Xin W, Zhi C, Shuhao LQ. An optimized method for automatic test oracle generation from real-time specification. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05) 2005. p. 440-9.
Xiong2012	P. Xiong RLP, B. Stepien. An Efficient Formal Testing Approach for Web Service with TTCN-3. 2012.