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: 11 of October 2012

Document version no.: 1.0

This document will be treated as strictly confidential. It will not be disclosed to anybody not having signed the ITEA 2 Declaration of Non-Disclosure.

**History**

|  |  |  |
| --- | --- | --- |
| Document version # | Date  (yyyy/mm/dd) | Remarks |
| 1.0 | 2012/10/11 | The first 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 |
| Robertas Jasaitis | KTU | jasaitis.robertas@gmail.com |
| Mantas Mitė | KTU | maantinis@@gmail.com |
| Eduardas Bareiša | KTU | eduardas.bareisa@ktu.lt |
| Dominykas Barisas | KTU | dominyko@gmail.com |
| Virginija Limanauskienė | KTU | virginija.limanauskiene@ktu.lt |
|  |  |  |
|  |  |  |

**TABLE OF CONTENT****S**

1. Introduction 4

2. Background 4

2.1. Model-based testing (MBT) 4

2.2. Model-Based GUI Testing (MBGT) 5

2.3. MBT in product line testing 6

3. Domain-specific test construction 6

3.1. Model-based GUI testing of smartphone applications 7

4. Automated testing of configurable systems 7

5. Test suite evaluation and optimization techniques 9

5.1. When can test-suite optimization be applied? 10

5.2. Test Suite Minimization 10

5.3. Test Suite Selection 11

5.4. Test Suite Prioritization 12

5.5. Optimal Test Suite Generation 12

6. Test automation for new development methods and processes 13

7. Search based software testing 13

7.1. Structural testing 14

7.2. Functional testing 15

7.3. Temporal testing 15

7.4. Assertion and exception testing 15

7.5. Challenges in SBST 16

7.5.1. Testability transformation 16

7.5.2. State-based programs 17

7.5.3. Execution environment 17

7.5.4. Oracle cost 17

8. Complex test data generation 18

8.1. Approaches for test data generation 19

9. Automated testing of distributed systems 21

10. References 33

# Introduction

This document is a work in progress that aims to present to ATAC partners and the European industry the state of the art in test automation. 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.

# Background

In a typical commercial development organization, the cost of providing assurance that the program will perform satisfactorily in terms of its functional and nonfunctional specifications within the expected deployment environments via appropriate debugging, testing, and verification activities can easily range from 50 to 75 percent 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 for example in unit and component testing.

## Model-based testing (MBT)

Model based testing (MBT) is a testing paradigm in which models are used to automatically drive the choice and sometimes the evaluation of test cases [Utting2006]. MBT aims at generating test cases from models describing some relevant aspect of SUT behavior at a chosen abstraction level. MBT is expected to allow more adequate software testing because it is rooted in automated procedures, avoiding error prone manual activities [Santos-Neto2007].

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 [Utting2011]. Some may be better suited in their supported modeling 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.

## 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. 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.

MBGT has the same challenges as MBT in general – manually crafting the models requires a lot of effort and deep expertise in formal modelling. The specifications have rarely enough details for creating the models, and mappings between the separately created models and the implementation have to be provided to be able to generate executable test cases. One way to reduce the required effort is using a reverse engineering approach to extract part of the model automatically [Grilo2010]. Another popular research topic is using some kind of formal language for writing the requirements and generating the models from the requirements, as in [Bertolino2010]. Especially in GUI applications, the requirements are often specified by people lacking technical expertise, so it is difficult to force them to use formal languages for specifications.

## MBT in product line testing

Verifying product lines with 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. 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].

# Domain-specific test construction

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.

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.

There are varying reasons for the complexity of the system. The system can be large, distributed, configurable or 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.

## Model-based GUI testing of smartphone applications

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 deal directly 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). MBT can be a considerable help in testing these interactions: if the applications are modeled individually using a suitable formalism and taking into account the shared resources they need, the individual models can then be combined into a single test model which can be used to test the concurrent execution of all the applications.

Also, the turnover of smartphone models on the market is rapid, and new products need to be tested quickly and efficiently. One way to reduce the efforts required to test new products is to reuse the artifacts (in MBT, primarily models) created for testing earlier products of the same product line. This is possible because the functionality of the applications usually changes very little between the products. In contrast, the GUIs of the applications are far more volatile, often necessitating significant updates to models. As a result, it is advantageous to separate the models for functionality from those for its implementation. The former models can then be reused for various products with minimal or even no changes and only the latter ones have to be designed anew.

These solutions naturally lead to the concept of a model library [Jääskeläinen2011], a collection of model components from which the actual test model can be composed. An individual model component depicts a small part of the system under test, such as the functional aspects of creating a multimedia message in the Messaging application. In order to generate tests, the appropriate model components are first selected from the library; for example, one could pick all components depicting the functionality of the Messaging and Contacts applications, plus those depicting their GUI on the product to be tested. These components are then composed into a test model which can be used to generate tests for the interaction of the desired features on the desired product.

# Automated testing of 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.

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 this year 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) research in testing of SPL is very little 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 this project proposal. If we consider the current situation of this year, still many publications in important research venues lack empirical studies on actual industrial systems, as for example [Uzuncaova2010], [Perrouin2010], [Cabral2010], [Mccaffrey2010].

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].

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.

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.

# Test suite evaluation and optimization techniques

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 necessary. 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 therefore 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 in a systematic way the test suite 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 case 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 redundancy 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.

## When can test-suite optimization be applied?

Given a suitable test case generation technique, one can easily start generating test cases. One of the major problems that appear during model-based testing is the large number of generated test cases, including a large amount of redundancy (e.g. identical test cases). This can result in problems when testing resources are missing or limited. In this task we consider three possible approaches in order to optimize test suites with regard to different aspects. We consider test suite minimization, test suite selection, test suite prioritization, and optimized test suite generation. Initial results from a partnership between industry and academy at transferring these optimization techniques into industrial practice are needed in order to fully understand the scenario of optimization and increase the efficiency and effectiveness of testing.

## Test Suite Minimization

It may be the case when testing resources are limited and is not possible to execute all test cases. Therefore researchers are considering test suite minimization or selection, which 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].

## 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 most 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.

## 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.

## 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 only 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.

# Test automation for new development methods and processes

In developing highly-complex systems, the use of new and evolving development methods, paradigms and processes, such as model-driven software development, continuous integration and agile processes, is becoming more common. For systems developed in this way, test automation is usually not readily available on all levels. In particular, there are no readily-available tools and methods for the automated testing of non-functional properties, nor are there automated testing methods for the systems developed. The goal is to extend existing methods in such a way that these kinds of systems can be tested automatically in an efficient way.

# Search based software testing

Search-based software testing 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.

In this project we will focus on applying search-based testing methods in two different complex embedded systems: Ericsson's telecoms platform, and Bombardier's rail vehicle control system. Our initial aim is to use search-based methods for test data generation for structural testing. We will also look into subsystems where functional, temporal or other non-functional testing is relevant – and use search-based testing for those purposes. Component level, integration, and system level testing will be investigated. Subsystems with different characteristics, and different levels of testing serve as case studies, where we will apply search-based methods.

The goal of the project is to identify and overcome the applicability problems of search-based testing at different testing levels, and for different types of testing purposes, in the context of the aforementioned complex industrial embedded software. Furthermore, we will investigate ways to combine other approaches with search-based testing, such as model-based methods, in order to create hybrid methods that overcome limitations of pure methods.

Search-based software testing (SBST) methods are shown to perform well on small examples of open-source software. Applying SBST on industrial-level code is an under studied topic. 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 [miller1976automatic], 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 [korel1990automated] 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

## 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. [wegener2001evolutionary]. Consider three nested if statements, where the search is trying to reach the True branch of the innermost conditional, as seen in Figure [fig:Approach-level]. Approach level shows how many more conditionals need to be satisfied to reach the target. Assume 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.

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 (count4) is not satisfied, then the branch distance would be 4-count+K. Branch distance formula depends on the type of the relational predicate. Tracey et al. [tracey1998anautomated] provides the full list of corresponding formula for different relational predicates (Figure [fig:Branch-distance-formula]).

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--x, =1.001. Arcuri [arcuri2010itdoes] discusses how different normalization formula may affect the search.

## Functional testing

DaimlerChrysler's automated parking system [buehler2003evolutionary, buhler2008evolutionary] 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 maneuver. 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.

## 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 has 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 [wegener1997testing, puschner1998testing]. As opposed to the static timing analysis, search-based methods under-estimates the WCET or over-estimates the BCET. Static analysis tend 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 [puschner1998testing, mueller1998fourthieee].

## 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 needs to be satisfied at that point in the code. Some assertions can be added automatically, such as division by zero. Korel and Al-Yami [korel1996assertionoriented] 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. [tracey2000automated] 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.

## Challenges in SBST

### 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. [harman2002improving2] 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. [mcminn2005testability] 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, do 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 (section [sub:Structural-testing]) 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. [baresel2003structural] 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 [tonella2004evolutionary] 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 [ferguson1996thechaining]. The chaining approach is based on analyzing 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 [mcminn2004hybridizing].

### 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 behavior 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 [mcminn2011searchbased].

### 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. [harman2010optimizing] 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 [fraser2011itis] discuss and suggest ways to overcome the problem of test length abnormally growing over time, for testing object oriented software. Lietner et al. [leitner2007efficient] present a combination of static slicing and delta debugging to efficiently minimize unit test cases.

# Complex test data generation

Object-oriented software is composed of various objects and their interrelations, there objects are defined by classes, possible/required relations between classes, and methods that could be performed by objects. It is very rare to find object-oriented software that would use only simple data types (integers, floats, strings, etc.) as methods arguments, in most cases the objects interact with each other by exchanging messages and passing other objects as arguments. A test data generator generates input values for the selected software unit. The selected unit is a method in a class. The method accepts some input values. The accepted input values are defined as method parameters. Each method parameter has a defined type. The tests generator has to create some values for each method parameter. 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. If the parameter type is float, the generated value modulus has to be within the range 1.04 \* 10-45 and 3.4 \* 1038 and can be either positive or negative. The value generation algorithms have to be different, one algorithm is used for generating value for parameters of the type signed short int, the other one has to be used for generating values for parameters of the type float. If the parameter is of type float or signed short int, the generation algorithm is straightforward – the generated value has to be selected from the allowed range (the range is determined by software under test implementation programming language and parameter type). But besides the numeric types there are more complicated types. One of them is a string. The type string defines a text of any length. The parameter of type string can be any length (but usually is limited by software under test programming implementation language, addressable memory space). The generator should select some string length and then generate a selected number of characters. The character generation algorithm could be similar as number generation – just selecting a one value from allowed alphabet.

The generation based on the values selection from allowed intervals algorithm is suitable for generating values for types which are built-in into programming language. The usual types are: integer, float, long, double, short, byte, char and string. But software implementations use complex types in most cases. The complex types are: arrays, lists, classes and structures.

The usual value generation algorithm for parameter of type array or list is simple: select an array or list length (the integer value generation algorithm can be used), generate a selected number (list/array length) of elements. For each array/list element generation the algorithm based on element type has to be used.

The test data generation for parameter of type class or structure is more complicated. The class or the structure for test generator can be viewed as a same entity. This entity is composed of other simple or complex types. The entity types can be composed of simple or complex types as well. To generate tests for method parameter which is of complex type (a class) test generator have to use data flattening technique [Meyer1997]. 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 8.1 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 8.1. 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 tex. But this approach has removed the meaning of a triangle.

## Approaches for test data generation

The simplest test data generation is random test data generation. 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. In this case test data contain mostly some meaningless data (from software domain perspective).

The authors are proposing modifications to random generation technique by employing feedback analysis from tests execution [Pacheco2007, Patrice2005]. Tests are executed and the coverage is calculated after each test execution. Based on the coverage level the decision is made if next random values have to be generated.

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. During the software under test unit analysis, its control flow chart is created. Based on graph theory methods the test inputs are generated, selected inputs drive the software execution by some paths. The main part of generator is input data selection which would force the code to be executed in the selected code branch. To achieve this, the constraints slowing techniques [Gotlieb1998], the relaxation method is used [Gupta1998]. These methods select initial values and based on the software execution feedback perform input values tuning. Unfortunately these approaches only work with values inside the unit under test which are of a simple type (float, integer, etc..) and are not capable to handle units which are calling other methods, functions and/or operates with variables of complex types (arrays, pointers, data structures, etc..). Authors have proposed some methods for tests generator when software uses pointers [Gotlieb2005], calls procedures and/or functions [Sy2003, Korel1996]. Tests generation, for software which uses complex data types, authors are proposing to employ data transformation into equivalent data types, for which existing generation methods are used [ 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.

The tests generation is based on the data gathered during software unit execution instead of data collected during code static analysis. The software unit is executed with some input data and during its execution runtime parameters are observed: executed paths, executed branches, executed operators. Based on observations the 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.

The genetic algorithms can be adapted for tests generation [Corno2004, Seesing2006, Harmanani2005, Kalpana2005]. The initial set of tests cases is created, after that tests are executed and theirs efficiency is measured by adhering the selected coverage criterion. During the next iteration child tests are generated by selecting better performing tests and killing less successful tests. Using this approach tests are created which achieve the selected coverage criterion with less test data or with less testing time. The advantage of these approaches is that tests generation is fast, the drawback – there is no guarantee that the defined tests generation goal will be reached at all.

Tests can be generated when the implementation of software under test is still not present. Tests are generated using software models. Formal and informal models can be used as a source for tests generation. It is also possible to generate tests directly from requirements specification, but that case the models are needed anyways, these models could be transformed from requirements specification, even from the textual ones [Gargantini1999]. During the tests generation using formal specifications the black box methods can be used, such as boundary values analysis and average values analysis. 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.

Formal specifications, 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 handy 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].

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.

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. The authors have proposed path finding tool [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. Then 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.

There are also methods for combining both techniques together [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].

# 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 threw an application. This is also valid for concurrent (multithreaded) applications as displayed by Figure 9.1. 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.

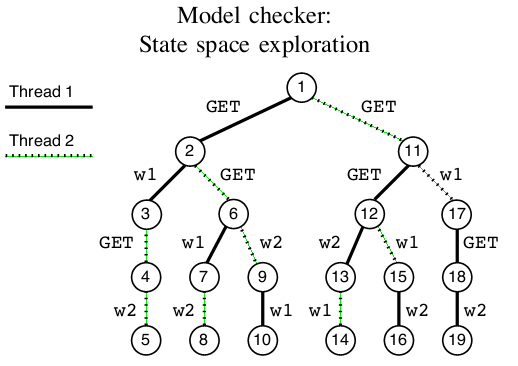
****

Figure 9.1. 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 9.2. 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.

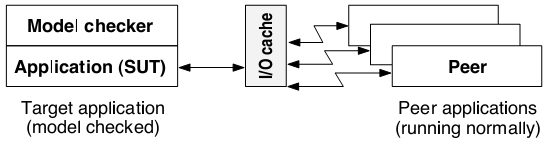
****

Figure 9.2. 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 9.3.

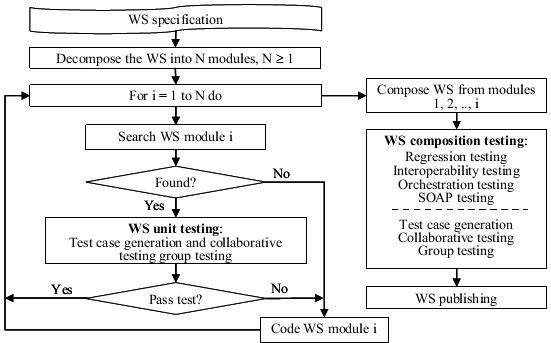
****

Figure 9.3. 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 9.4.

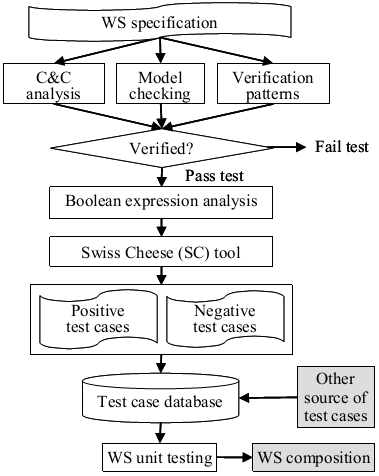
****

Figure 9.4. 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 9.5. 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].

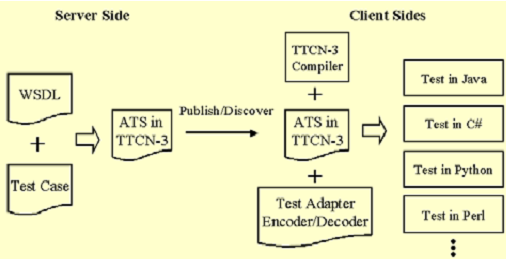
****

Figure 9.5. 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. Authors 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, preconditions and post conditions of class methods. Authors give banking class diagram as an example which is shown in Figure 9.6. 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].



Figure 9.6. 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 9.7.



Figure 9.7. 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 9.8.

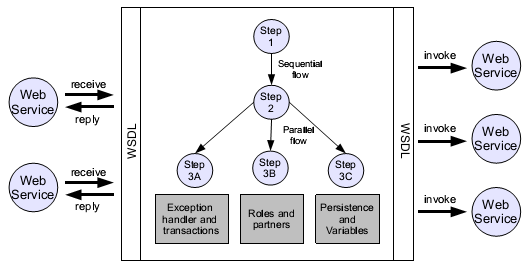
****

Figure 9.8. BPEL composition [Endo2008]

Graphical WSC example called Loan process presented by Figure 9.9.



Figure 9.9. 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 9.10 represents PCFG of Loan Process Example.



Figure 9.10. 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.

# References

|  |  |
| --- | --- |
| Utting2006 | M. Utting and B. Legeard, “Practical Model-Based Testing: A Tools Approach”, Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 2006. |
| Utting2011 | Utting, M., Pretschner, A., Legeard, B., “A Taxonomy of Model-Based Testing Approaches”, Software Testing, Verification and Reliability, 2011. |
| 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. |
| 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. |
| 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. |
| Vieira2008 | M. Vieira et al., "Applying Model-Based Testing to Healtcare Products: Preliminary Experiences," in Proceedings of the 30th International Conference on Software Engineering (ICSE08), Leipzig, Germany, 2008, pp. 669-671. |
| 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 |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| Halpern2002 | B. Hailpern and P. Santhanam, “Software debugging, testing, and verification”, IBM SYSTEMS JOURNAL, VOL 41, NO 1, 2002 |
| Hinchey2012 | M. Hinchey and L. Coyle, “Conquering Complexity”, 466 pages, Springer |
| Meyer1997 | Meyer B. Object-oriented software construction: Prentice-Hall, Inc.; 1997. |
| Duran1984 | Duran JW, Ntafos SC. An evaluation of random testing. IEEE transactions on software engineering 1984;10:438-44. |
| 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. |
| 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 |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| Korel1996 | 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. |
| 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. |
| Beyer2004 | Beyer D, Chlipala AJ, Majumdar R. Generating tests from counterexamples. 26th International Conference on Software Engineering (ICSE'04) 2004. p. 326-35. |
| Ferguson1996 | Ferguson R, Korel B. The chaining approach for software test data generation. ACM Trans Softw Eng Methodol 1996;5:63-86. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| Uhl2003 | Uhl A. Model driven arcitecture is ready for prime time. Software, IEEE 2003;20:70, 2. |
| Mellor2003 | Mellor SJ, Clark AN, Futagami T. Model-driven development - Guest editor's introduction. Software, IEEE 2003;20:14-8. |
| Spivey2008 | Spivey JM. Understanding Z: A Specification Language and Its Formal Semantics: Cambridge University Press; 2008. |
| Packevičius2006 | Packevičius Š, Kazla A, Pranevičius H. Extension of PLA Specification for Dynamic System Formalization. Information Technology and Control 2006;3:235-42. |
| 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. |
| Fowler2003 | Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Third Edition ed. Boston: Addison-Wesley Professional; 2003. |
| 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. |
| 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. |
| Clark2002 | Clark T, Warmer J. Object Modeling with the OCL: The Rationale behind the Object Constraint Language (Lecture Notes in Computer Science): Springer; 2002. |
| 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. |
| 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. |
| 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. |
| 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. |
| Hessel2006 | Hessel A, Pettersson P. Model-Based Testing of a WAP Gateway: an Industrial Study. FMICS and PDMC2006. |
| Artho2009 | C. Artho WL, M. Hagiya, Y. Tanabe, M. Yamamoto. Cache-based Model Checking of Networked Applications: From Linear to Branching Time. ACE 2009. |
| 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. |
| 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. |
| Tsai2003 | W.T. Tsai LY, Feng Zhu and Ray J. Paul. Rapid Verification of Embedded Systems Using Patterns. COMPSAC 2003:466-71. |
| 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. |
| Xiong2012 | P. Xiong RLP, B. Stepien. An Efficient Formal Testing Approach for Web Service with TTCN-3. 2012. |
| Brucker2001 | A. D. Brucker BW. Testing Distributed Component Based Systems Using UML/OCL. IEEE 2001:608-14. |
| 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. |
|  |  |
|  |  |