D 7.1: Evaluation of existing methods and principles

Fabrice Bouquet, Frederic Dadeau, Stéphane Debricon, Pierre-Alain Masson (INR), Zoltán Micksei, Daniel Varro (BME), Berthold Agreiter, Michael Felderer (UIB), Bruno Legeard, Julien Botella, Olivier Bussenot, Eddie Jaffuel, Christophe Grandpierre, Jean-Luc Hamot, Aurélien Masson (SMA), Elisa Chiarani, Federica Paci, Fabio Massacci (UNITN), Jan Jurjens (OU/TUD)

Document Information

Document Number	D7.1.
Document Title	Evaluation of existing methods and principles
Version	6.4
Status	Final
Work Package	WP 7
Deliverable Type	Report
Contractual Date of Delivery	M12.
Actual Date of Delivery	16 June 2010
Responsible Unit	INR
Contributors	INR, UIB, SMA, BME, TUD, UNITN
Keyword List	Model-based, Test
Dissemination	level PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	09.06.26	Draft	F. Bouquet (INR)	First version
0.2	09.07.2	Working	F. Dadeau (INR)	Paragraph 4.1
0.3	09.07.2	Working	PA. Masson (INR)	Paragraph 3.2
0.4	09.07.22	Working	Z. Micksei (BME)	Section 5
1.0	09.07.22	Draft	B. Agreiter (UIB)	Review
1.1	09.07.27	Working	D. Varro (BME), E. Chiarani, F. Paci (UNITN)	Quality check
2.0	09.07.29	Working	B. Legeard (SMA)	Review
2.1	09.07.29	Working	J. Jurjens (OU/TUD), E. Chiarani (UNITN)	Quality check
3.0	09.07.30	Draft	F. Bouquet (INR)	Review
3.1	09.07.30	Working	J. Jurjens (OU/-TUD), F. Massacci, E. Chiarani (UNITN)	Quality check
3.2	09.08.13	Working	B. Agreiter, M. Felderer (UIB)	Section 2.1 and review
3.3	09.08.23	Working	F. Bouquet (INR)	Review
4.0	09.08.25	Draft	F. Bouquet, PA Masson (INR)	Review
4.1	09.08.25	Working	E. Chiarani (UNITN)	Quality check
5.0	09.08.26	Draft	F. Bouquet (INR)	Minor edits + revised version
5.1	09.08.26	Working	E. Chiarani (UNITN)	Minor edits + revised version
5.2	09.08.31	Working	F. Bouquet (INR)	Minor edits
6.0	09.08.31	Final	J.Jurjen (OU/TUD), E. Chiarani (UNITN)	Finalised version
6.1	10.06.10	Revised Version	F. Bouquet, F. Dadeau, S. Debricon, P. Masson (INR)	Revised version
6.2	10.06.11	Revised Version	B. Legeard, J. Botella, O. Bussenot, E. Jaffuel, C. Grandpierre(SMA)	Revised version
6.3	10.06.15	Revised Version	E. Chiarani (UNITN), B. Agreiter (UIB)	Quality check completed, minor changes in the formatting
6.4	10.06.16	Final Version	S. Debricon (INR)	Final Version

Executive summary

The objective of Work Package 7 (later on WP7) is to ensure the preservation of security of long-life evolving systems by means of model based tests.
Model-Based Testing (MBT) involves the automatic derivation of test cases, in whole or in part, from a model that describes at least some of the aspects of the System Under Test (SUT). Therefore, MBT is the automation of a black-box test design. A MBT tool uses various algorithms and strategies to generate tests from a behavioral model of the SUT. Such a model is usually a partial representation of the SUT's behavior, partial because the model abstracts away some of the implementation details. Test cases derived from such a model are functional tests on the same level of abstraction as the model. The generated test cases are grouped together to form an abstract test suite (or test repository). Although model-based testing has a high potential of reducing test costs and increasing test quality, this technique is adopted slowly in industrial practice. This is in particular due to the important question of managing changes and evolutions in the testing process.

Key issues facing SecureChange WP7 are to address the life-cycle of an automatically generated test repository, dedicated to test security aspects of the SUT, for evolving systems.
This document provides some criteria to evaluate model-based testing approaches with respect to evolution and security of systems. We present a survey of the state of the art of MBT methods and introduce the background technology that constitutes the starting point of the project.

This document consists of four parts. The first chapter concern the analysis of MBT challenges for evolving systems. We introduce several criteria to evaluate existing MBT methods in the context of automated testing of evolving systems for security requirements.
The second chapter is a state of the art of MBT approaches, including a taxonomy based on their characteristics.
The third chapter give a technical background of the WP7 partners that will be used to address SecureChange challenges.
The last chapter is the summary of identified key issues that will be resolved by the project.

Index

List of Figures

List of Tables

Abbreviations and Glossary

Abbreviations

Abbreviations	**References**
API	Application Programming Interface
FSM	Finite State Machine
ISTQB	International Software Testing Qualifications Board
MBT	Model-Based Testing
REQ	Requirement
SUT	System Under Test
TTS	Telling TestStories

Table 1: Abbreviations used in the document

Glossary

Term	Definition
Adapter	Piece of code to concretize logical tests into physical tests
Evolution Test Suite	Test Suite targeting SUT evolutions
Logical Test	See Test Case
Model Layer	Link of model's operations in Test Scenario
Model-Based Testing	Process to generate tests from a behavioural model of the SUT
Status of Test Case	new, obsolete (outdated, failed), adapted, reusable (reexecuted, unimpacted)
Physical Test	See Test Script
Requirements	Collection of functional and security requirements
Regression Test Suite	Test Suite targeting non-modified part of the SUT
Stagnation Test Suite	Test Suite targeting removed part of the SUT
System Model	Model of the SUT used for development
Test Case	A finite sequence of test steps
Test Intention	User's view of requirement into Test Scenario
Test Model	Dedicated model for test capturing the expected SUT behaviour
Test Suite	A finite set of test cases
Test Script	Executable version of a test case
Test Scenario	A test generation strategy
Test Sequence	See Test case
Test Step	Operation's call or verdict computation
Test Strategy	Formalization of test generation criteria
Test Objective	High level test intention

Table 2: Glossary

1. Challenges of MBT for evolving systems

This chapter presents the challenges for model-based testing approaches in evolving systems. The first section presents key challenges introduced by testing security aspects of long-life evolving systems. The second section defines criteria to evaluate existing Model-based testing methods in this context.

1.1 Problems Statements

Model-based testing renews the whole process of functional software testing: from business requirements to the test repository, with manual or automated test execution. It supports the phases of designing and generating tests, documenting the test repository, producing and maintaining the bi-directional traceability matrix between tests and requirements, and accelerating test automation. Those approaches are a field of active researches from at least the mid-90s leading to commercial solutions such as Conformiq Qtronic, Microsoft SpecExplorer or Smartesting Test Designer.

Model-based testing (MBT) is an increasingly widely-used technique for automating the generation and execution of tests. There are several reasons for the growing interest in using model-based testing:

- The complexity of software applications is constantly increasing and the user's aversion to software defects is greater than ever. Due to this, functional testing has to become more effective at detecting bugs;
- The cost and time of testing is already a major proportion of many projects (sometimes exceeding the costs of development). There is a strong tendency to investigate methods like MBT that are able to decrease the overall cost of testing by the automatic derivation as well as execution of tests.
- The MBT approach and the associated commercial and open source tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can provide a good ROI [81].

Long-life evolving systems introduce specific challenges that are not supported by current approaches. Those challenges are:

- Test repository maintenance and management with evolving requirements. In that case a full re-creation of the repository after each evolution is not acceptable. The testing team needs a stable repository to ensure a continuous validation process. The team must keep track of any modification to the test repository issued by the fault analysis on the System Under Test (SUT).

- Impact analysis on the test repository should provide a test classification for the system after evolution steps (regression, evolution, stagnation tests).
- MBT security testing has been a vast field of research particularly in the context of access control policies. But there are no previous work on testing security properties of evolving systems.
- Tracing requirements in a MBT approach for testing security properties of evolving systems.

Those challenges will be addressed by WP7 partners. The following section will provide a list of evaluation criteria that will allow the characterization of MBT approaches with respect to those challenges.

1.2 Evaluation criteria for existing methods

Name	Description	Evaluation
Stability of test repository	Ability to minimize the impact of evolutions on the test repository in term of creation / deletion of tests	scale 1..3 (1: complete re-creation, 3: maximum tests re-use)
Traceability of changes	Ability to trace an evolution from requirements to test repository	scale 1..3 (1: no traceability, 3: full traceability)
Impact analysis	Ability to inform the user on potential impacts of an evolution on the test repository	scale 1..3 (1: no impact analysis, 3: full impact analysis)
Test suite qualification based on changes	Ability to create test suite based on change type	qualification / no qualification

Table 1.1: Evaluation criteria for change

In [83], Utting and al. propose a general taxonomy of MBT approaches. This paper defines six characteristics based on the classical MBT life-cycle (modeling, test generation, test execution). However, those characteristics does not provide dedicated evaluation criteria for MBT applied to long-life evolving systems.

Name	Description	Evaluation
Traceability of security properties	Ability to provide bi-directional traceability between security properties and generated test cases managed in the test repository	scale 1..3 (1: no traceability, 3: full traceability)
Completeness of security testing	Ability to manage the test of functional security properties and to find security vulnerabilities (threats and attacks)	both / functional security properties only / security vulnerabilities only

Table 1.2: Evaluation criteria for security

In the context of the SecureChange project, we propose six new criteria to be used to evaluate MBT approaches defined on two dimensions that are:

- Change: criteria to evaluate how MBT methods deal with evolutions in the context of long-life evolving systems presented in Table 1.1. There are four criteria associated with Change to take into account evolution all around the testing process.
- Security: criteria to evaluate how MBT methods deal with security properties in the context of long-life evolving systems presented in Table 1.2. There are two criteria associated with Security to establish confidence in the generated specific security tests.

In section 4.1, those criteria are used to compare several MBT methods to test long-life evolving systems. The following chapter proposes a state of the art of MBT approaches.

2. State of the Art

We base our study of the state of the art on a guideline. This guideline is a taxonomy of Model-Based Testing methods proposed by Utting and al. [83]. We can analyze each existing approaches to compare them. We identify elements that help our approach to take into account evolution and security of evolving systems.

The following two sections of this chapter first describe the taxonomy, and then present the existing approaches which we have taken into account.

2.1 Taxonomy of MBT approaches

We present in this section a taxonomy of the Model-Based Testing approaches [9], that provides an overview of the state of the art. This taxonomy is extracted from [83].

2.1.1 Overview of the Model-Based Testing Process

The global process of Model-Based Testing is depicted in Fig. 2.1. It is composed of 5 main steps, that are now described.

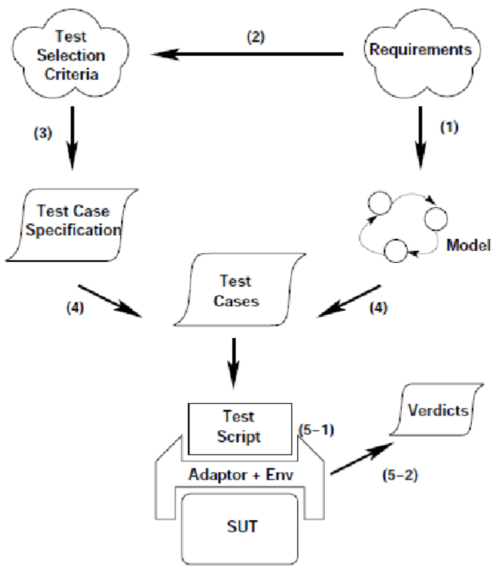


Figure 2.1: The Model-Based Testing Process

1. The first step consists in designing a formal model from the initial requirements (informal). It is mandatory that this model takes into account the (functional, security, etc.) requirements that the system must fulfill.

2. The second step consists in choosing a test selection criterion that will make it possible to exploit the model content so as to produce the test cases. These criteria may aim at: the coverage of a model functionality, the coverage of the model states/data, a random walk in the model, highlighting a given set of faults, etc.

3. The third step is the formalization of the test cases, which gives a high level description of the test cases that one wants to generate. It represents an intermediate step, that formalizes the test selection criterion so as to make it operational (to be automatically applied on the model).

4. The fourth step consists in the generation of the test cases themselves. These tests are said to be abstract since they are expressed at the model level, using the model entities (data and operations). This step can be automated by a *test generator* that is able to deal with test case formalization produced at step 3. The resulting abstract test cases are made to fulfil the chosen test selection criterion.

5. The fifth and final step consists in executing the tests on the System Under Test (SUT) and assigning the test verdicts. This is usually done in two steps.

   5–1 **Concretization of the abstract tests.** This substep consists in translating the abstract test cases into concrete calls to the SUT interfaces.

   5–2 **Verdict assignment.** This substep consists in interpreting the results obtained from the SUT in response to the stimuli. First, these results are translated back into the formalism of the model, in order to be compared to the expected results that were computed from the model. A conformance relationship is used to check that the results conform to each other.

   Notice that, in case of online testing (i.e. when tests are generated and played at the same time on the SUT), this step is coupled with step 4.

2.1.2 Taxonomy of the MBT Approaches

We briefly review the classification proposed in [83]. This classification takes into account three main aspects, that are decomposed into subcategories:

- the model, that is seen as a scope, the characteristics of the modelled elements, the modelling paradigm,
- the test generation technique, involving test generation criteria and an underlying test generation technology,
- the execution of the tests, that can be either online (tests are generated and played on the SUT at the same time), or offline (tests are generated and stored in a test repository, before being translated to be run subsequently on the SUT.

We now detail the first two aspects.

Model

As explained previously, the models are classified according to three criteria.

Scope The model scope defines whether the model is input-only, or input-output.

Input-only models are models that only specify valid inputs for the system without providing any information on the expected outputs. Such models can be used to generate test data (but, not necessarily test cases). However, they can not be used for conformance testing. Nevertheless, these kinds of models can notably be used when performing robusteness testing, when the test objective is to ensure that the system does not crash.

On the contrary, input-output models specify both valid inputs and expected outputs which makes them a suitable choice for conformance testing.

Characteristics The model characteristics answer the three following questions:

- Is the model timed or untimed?
- Is it a determinstic or non-deterministic model?
- What is the dynamics of the model: continuous time, discrete time, or both ?

Modelling paradigms The modeling paradigms are the following.

- State-based notations: these notations describe a system using a set of state variables and operations modifying them. Each operation is specified using preconditions (describing the licit uses of the operations) and postconditions (describing the effect of the operation). Some state-based notations: B [1], Z [76], VDM [55], JML [16], UML/OCL [73, 85].
- Transition-based notations: these notations describe the transitions between the states of the system. These are usually graphical notations, using nodes to represent states, and edges to represent transitions. These formalisms can be extended to take into account state variables, guards on the transitions, etc. Some transition-based notations: finite state machines (FSM) [63], (input-output) labelled transitions systems [80], statecharts [46].
- History-based notations: these notations describe a system through its acceptable traces. Such notations make it possible to address different notions of time (discrete or continuous, linear or tree-based, etc.) The message sequence charts (MSC) [47] formalism falls into this category.
- Functional notations: these notations describe the system using a set of mathematical functions (first order functions for algebraic specifications [41]), or of higher order (as in HOL).
- Operational notations: these notations describe the system as a set of executable processes that are run in parallel. These notations cover the process algebra, CSP, CSS, or Petri Nets [70].
- Stochastic notations: these notations describe the system using a probabilistic model of the events and input data. They are mainly used for modelling the SUT environment rather than the SUT itself. For example, Markov chains can be used to model operational profiles of the SUT.
- Data-flow notations: these notations are focused on the data flow rather than the control flow. Lustre [68] or Matlab Simulink [11] are classified into this category.

It is worth noticing that some notations are not restricted to a single modelling paradigm, e.g. UML combines a state-based formalism (using OCL constraints) and a transition-based formalism (using the statecharts diagrams)

Test Generation

The test generation aspects rely on two elements: the test selection criteria that can be applied, and the test generation technology that is used to compute the test cases.

Test Selection Criteria The considered test selection criteria are the following:

- Structural model coverage: this criterion depends on the modelling formalism that is used. On pre/postconditions models, it aims at covering the cause-effects, the disjunctions in the preconditions, etc. On transition based models, it aims at covering the entities of the automaton (states, transitions, etc.). On textual models, some of the structural testing criteria can be revisited and adapted to cover the code of the model.
- Data coverage: it represents the test data selection strategy, that can be very simple (random values), or more complicated (e.g. involving pairwise testing or a boundary value analysis).
- Requirement coverage: this criterion uses the requirements (functional or security requirements) to target specific parts of the model (e.g. states, transitions, transitions sequences, etc.) to be exercised.
- Test cases specification: this criterion consists, for the user, to provide a test scenario that he wants to execute on the model. This scenario can be expressed in a textual way[32, 22], or using an automaton [54].
- Random or stochastic criteria: these criteria aim at the coverage of an operational profile of the system. Different probabilities of usage are associated to different parts of the model. The test generation process will produce tests accordingly to these probabilities.
- Fault-based criteria: these criteria aim at using a fault model that has to be covered by the test generation approach, meaning that the tests are produced so as to detect all the faults that are provided in the fault model.

Test Generation Technology The test generation technology is the technique that is used to generate the test cases. We consider six technologies.

- Random generation: the test cases are computed in a random manner for selecting the operations and their associated inputs' values.
- Search-based algorithms: this technique uses specific graph algorithms (e.g. the rural chinese postman), that aim at achieving the selected test selection criterion. Such algorithms also include genetic algorithms.
- (Bounded) Model Checking: this technique consists in an exploration of the state space of described by the model, using a model-checking tool. This techniques usually involves the use of a property (invariant or temporal) that is given as an input to the model checker for producing a trace that can be used as a test case.

- Symbolic execution: it consists in simulating the execution of the system (or model) by replacing the input values by symbolic variables.
- Theorem proving: this technique consists in using a a theorem prover to produce the test cases.
- Constraint solving: it similarly consists in using constraint solving techniques to generate test data, or test cases, coupled with search algorithms.

Take into account that some tools may combine several test generation technologies in order to produce the test cases.

We now review some well-known approaches of the literature.

2.2 Existing Approaches

An issue of model-based testing is to be able to measure and insure coverage from test to the model. The intention guarantees only test coverage on model and not on the SUT. We identified two kinds of criteria to insure coverage.

2.2.1 Static Criteria

Static criteria are based on two coverage approaches: control flow oriented and data flow oriented.

Control flow graph criteria

A method for structural testing based on the control flow coverage is to propose a certain set of paths in a graph in order to form test campaigns. Satisfying a structural testing method for a given coverage criterion is therefore to find tests that enhance control paths (i.e. paths of execution) and covering paths provided by the method adopted. There are several coverage criteria based on graph control:

- Coverage of all-nodes or statement coverage,
- Coverage of all-arcs or decision coverage,
- Coverage of path and internal boundaries,
- Coverage of all i-paths,
- Coverage of all paths.

This list is ordered from the lowest to the highest criterion to test (except 'coverage of paths and internal boundaries' which is not classifiable). In general, the stronger this criterion is, the higher is the number of test data to satisfy.

To cover criteria, a test suite must activate a dedicated part of specification as follows [82]:

- **State Coverage (SC)**: test suite must execute every reachable statement
- **Decision Coverage (DC)**: test suite must ensure that each reachable decision is made true by some tests and false by other tests. Decisions are the branch criteria that modify the flow of control in selection and interaction statement.

- **Path Coverage (PC)**: test suite must execute every path to satisfy through the control flow graph.
- **Condition Coverage (CC)**: test suite achieves CC when each condition is tested with a true result and also with a false result. For condition containing N conditions, two tests can be sufficient to achieve CC.
- **Decision/Condition Coverage (D/CC)**: test suite achieves D/CC when it achieves both decision coverage (DC) and condition coverage (CC).
- **Full Predicate Coverage (FPC)**: test suite achieves FPC when each condition is forced to true and to false in a scenario where that condition is directly correlated with the outcome of the decision.
- **Modified Condition/Decision Coverage (MC/DC)**: This coverage strengthens the directly correlated requirement of FPC by requiring the condition c to independently affect the outcome of the decision d. A condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible condition.
- **Multiple Condition Coverage (MCC)**: test suite achieves MCC if it exercises all possible combination of condition outcomes in each decision.

In [66], for code-based coverage we have PC > DC > SC, where C1 > C2 indicates that every test suites satisfies C1 also satisfies C2. More generally as propose in [82], Figure 2.2 gives hierarchy between criteria.

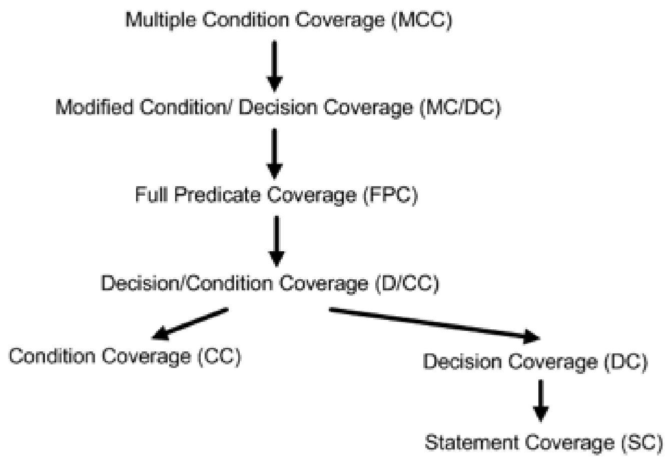

Multiple Condition Coverage (MCC)
Modified Condition/ Decision Coverage (MC/DC)
Full Predicate Coverage (FPC)
Decision/Condition Coverage (D/CC)
Condition Coverage (CC)
Decision Coverage (DC)
Statement Coverage (SC)

Figure 2.2: The hierarchy of control flow coverage criteria

In some case, we can define dedicated criteria as transition based coverage criteria as in Finite State Machines. These criteria are close to the previous criteria.

Data flow criteria

Data flow can be annotated with extra information regarding the definition and use of data variables. Informally, a definition of a variable is a *write* to the variable and a *use* of a variable is a read from it. For a given variable v, we say that (d, u) is a def-use pair if d is a definition of v and u is a use of v, and there is a path from d to u that is free to other definitions of v. So data flow criteria attempt to cover:

- **All-defs**: the all-definition criterion requires a test suite to test at least one def-use pair (d, u) for every definition d, that is, at least one path from each definition to one of its feasible uses.
- **All-uses**: the all-uses criterion requires a test suite to test all def-use pairs (d, u). This means testing all feasible uses of all definitions.
- **All-def-use-paths**: The all-def-use-paths criterion requires a test suite to test all def-use pairs (d, u) and to test all paths from d to u.

We have this hierarchy:

$$\text{All-def-use-paths} \rightarrow \text{All-uses} \rightarrow \text{All-defs}$$

We can define complementary criteria with external additional information on the model.

2.2.2 Dynamic Criteria

Dynamic criteria are about the sequencing of states or actions of the model. Several ways have been explored to express such criteria. For example in [6], the dynamic criterion is a sequencing of states expressed as a temporal logic (PLTL) property. It is a sequencing of actions expressed in the shape of an IOLTS in [17] and [52], or as a regular expression in [62]. We propose to describe a dynamic criterion, denoted as TP for Test Purpose, as a sequencing of states and actions. Its semantics is an automaton whose states are interpreted as state properties and whose transitions are labeled by action names. In our approach, the validation engineer manually describes by means of a test purpose TP (see Def. 1) how he intends to test the system, according to his know-how. We have proposed in [56] a language based on regular expressions, to describe a TP as a sequence of actions to fire and states to reach (targeted by these actions). States are described as state predicates. Actions can be given either explicitly, or under the generic name $op.

Definition 1 (Test Purpose) *A test purpose on a model M (with a set OM of operations) is a tuple* $< Q^P, {q_0}^P, T^P, \lambda^P, Q^P, QfP >$ *where :*

- Q^P *is a finite set of states,*
- ${q_0}^P$ *is an initial state,*
- ${Q_f}^P \subseteq Q^P$ *is a set of accepting states,*
- $T^P \subseteq Q^P \times (O^M \cup \{\$op\}) \times Q^P$ *is the set of labelled transitions,*
- $\lambda^P \subseteq Q^P \rightarrow Pred^M$ *is a total function that associates with each state q a state predicate denoted as* $\lambda^P(q)(\subseteq Pred^M)$.

In [57], we have presented a test generation approach from a TP and a B model, in which every action is described by an operation. This method proceeds by unfolding all paths of the automaton associated to the TP. Each path is a symbolic test, in the sense that the values of the operation parameters are not defined. They will be defined by an instantiation phase that uses constraint solving techniques and boundary valuation strategies. Also, the test must be concretized to become executable on the SUT. This approach has been successfully experimented on the industrial application IAS (Identification Authentication and Signature) with Gemalto.

Model-based testing and verification techniques

Research on model-checking has focused on the ability of these tools to fight the state space explosion and on increasingly expressive modeling paradigms and languages to express the property to be proved. Tools such as SPIN [49], Uppaal [61], CADP [33] have been developed to prove reachability, safety, liveness and fairness properties expressed in temporal logics on models in the form of communicating automata, timed automata models or process algebra. A recent evolution is the use of SAT or SMT [7] solvers to perform bounded model checking on infinite-state systems (SAL) [26], hybrid systems (HySAT [35], HyTech [48]) or Lustre (Prover).

Another direction is probabilistic model-checking with tools like PRISM [60] whose goal is to qualitatively or quantitatively evaluate the satisfaction of a property on a model.

Embedded systems have particular characteristics to be taken into account by V&V tools. They are often cyclical reactive systems that must be modelled using specialized paradigms such as synchronous languages, for which specialized tools such as Gatel [65] and Prover have been developed. A combination of continuous and discrete-event behaviour may need to be modelled, using the hybrid systems paradigm. This is treated by the HySAT model-checker but the research on testing of hybrid systems is a very recent research area [2]. They are usually real-time systems that are represented with models that use an explicit representation of time. Time properties can also be represented by a list of discrete events that take place. This is possible in the analysis tools based on timed automata or Petri Nets because timing properties are more susceptible to analysis than to test. The software used for embedded systems is often concurrent meaning that all possible inter-leavings of concurrent behaviour must be taken into account. Here again, analysis techniques are better developed than test, although there is research on test of concurrent systems [74]. Because of the close integration of software and hardware for embedded systems, software must often be tested in the target environment, introducing problems of injection of test-case values and observability of results. This has motivated the development of simulation languages such as SystemC [44] and methodologies to test embedded software with hardware in the loop. The model of computation introduced with the Signal synchronous language [10], then further developed in the Polychrony [45] workbench and industrialized in RT-Builder [72], consists in considering a median (polychromous or multi-clocked) model of computation into which heterogeneous specification can be interpreted, and from which sequential or distributed real-time code can be generated, used for analysis, simulation and test purposes.

2.2.3 Regression Testing

Changes in software artefacts throughout its lifecycle could make previously fixed bugs re-appear or brake existing functionality [12]; therefore systems should be retested after a modification is made. Changes can happen in subsequent development phases or after the software enters its maintenance phase. This retesting is usually referred as regression testing [9].

Regression testing is defined as "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or the component still complies with its specified requirements" [29]. Therefore, the intention is to test whether what was working before is still working, and previously fixed bugs do not reappear.

Regression testing can be performed on any testing level (i.e., module, integration, etc.), and it can cover both functional and non-functional requirements. However, rerunning every test after each of the modifications is not feasible, thus a trade-off must be made between the confidence gained from regression testing and resources used for it [51]. For this reason, several regression testing techniques were proposed over the years, e.g. to select only a subset of the regression test suite, what is relevant for the current change, or to identify those new parts of the system, which are not covered by existing tests.

Regression testing techniques

The research in the field of regression testing focused on the following problems:

1. Regression test selection: select only tests from the regression test suite that are affected by changes.
2. Test suite minimization: find a minimal subset of test cases that preserves the coverage with respect to a certain criterion of the original test set.
3. Coverage identification: identify those parts of the system that need additional tests due to the change.
4. Test prioritizing: optimize the order of tests according to some criteria, e.g. to run those tests first which are more likely to uncover bugs or which need less time to run.
5. Test suite execution: automatically execute the test in an efficient way.

For regression test selection techniques the basic idea is similar to the one used in build systems (e.g. the make tool), namely that at each build only those files need to be recompiled that have been changed or depend on a file that have been changed. Similarly, to reduce the size of the regression test suite, and thus reduce the time and resources needed to execute it, one can select only those tests that work on changed parts of the system. Rothermel and Harrold published a detailed survey paper about regression selection techniques [71]. They evaluated several techniques according to their inclusiveness, precision, efficiency and generality. The surveyed techniques consisted of linear equation, symbolic execution, path analysis, dataflow, program dependence graph, system dependence graph, modification based, cluster identification, slicing, graph walk techniques, etc. Each technique had its strength or weakness; some were able to uncover more errors, while some computed the selected tests very fast.

As the test suite grows and changes, some tests become redundant. Test suite minimization techniques remove test cases from the tests suite to retain only a minimal number of test cases, while providing the same level of coverage than the original test suite [53]. However, care must be taken, because removing too much test cases can reduce its fault detection effectiveness.

Changes in the system can introduce new parts, which are not exercised by existing tests. Coverage identification can map these parts of the system. Simple approaches can use code coverage analysis tools [88] to uncover changed portions not touched by existing tests. More advanced approaches typically use some sophisticated data structure, e.g. program dependence graphs [34] that capture also data and control dependencies in the source code.

Test prioritization techniques can have several goals. One can optimize the order of the test suite to increase the rate of fault detection, code coverage, or the rate at which high-risk faults are detected. Rothermel et al. analyzed in [28] nine test prioritization techniques (e.g. random, prioritize in order of coverage statements, etc). Their conclusion was that even simple approaches (which are quite easy to implement and inexpensive) can improve the rate of fault detection. However, the performance overhead of more sophisticated approaches was still a bit high.

Test suite execution techniques concentrate on the automatic execution and evaluation of test cases. These techniques moved into the practice over the years, as most of the current testing tools have these functionalities.

Tools for regression testing

Running a set of regression tests is usually part of the automatic build procedures of popular, modern software development processes. However, industrial testing tools and platforms used nowadays (both commercial [50, 69] and open source [27, 39] usually concentrate on just the automatic execution of tests, collection of results, and creating test reports when talking about regression testing. These tools usually do not use techniques presented in the previous section, they do not perform test selection or minimization on the regression test suite.

On the other hand, several academic tools were reported to support research on different regression testing techniques. The drawback of these tools is however that they are usually not available to the public or not maintained any more.

TestTube [20] was a tool developed at AT&T Bell Laboratories for selective retesting of C programs. It instruments the source code to capture which part of the system is covered by each tests, then computes which tests are needed for a given modification.

Automatic Testing Analysis tool in C (ATAC) [86, 75] combines modification-based test selection technique with test set minimization. It instruments the program when tests are executed. Using the recorded information and costs assigned to tests the tool can select tests that give maximal coverage, and later it can reduce this test set with respect to block coverage.

Echelon [77] was a tool developed by Microsoft Research for test case prioritization. It works on binary level to identify changes between the current and the previous version. Echelon uses a fast binary matching technique instead of expensive data flow analysis. The tool then prioritizes the tests according to the number of changed blocks they cover. Echelon also lists those blocks, which are not covered by existing tests. The scalability of this tool was tested using large binary production e.g. created for a project with one with 1.8 million LOC.

Model based regression testing

The previous listed approaches mainly work on source code. Instead of identifying the dependencies and effects of changes using code analysis techniques [23, 42, 43, 48], the analysis can be carried out on the model level. These methods have the advantage, among other things, that the models are usually smaller due to the operating on a higher abstraction level.

The approach presented in [19] generates regression test suites from Extended Finite State Machine (EFSM) models. A dependency analysis searches for the effects of changes expressed as elementary modifications (i.e. adding, deleting or changing transitions), and creates test cases for the changed parts of the system. The method presented in [84] works similarly on EFSM models, and its focus is to reduce an existing regression test suite based on dependency analysis.

2.2.4 Model-Based Security Testing

This section presents the use of the Model-Based Testing process in the context of testing the security of a system. This section is divided into two parts. The first one deals with a functional approach in which the security aspects of the system are embedded within the model and the subsequent test generation approach focuses on these aspects to exercise it. The second part is dedicated to approaches that consider a modification of the model that represents common attacks that can be performed on the system.

Functional approach

A part of the security requirements w.r.t. a system can be expressed as security properties. Here, we consider a context in which the security aspects of the system are embedded within the functional model. From the model, a set of security properties is formally expressed. These properties should hold on the model as it incorporates the security aspects. We do not address here the question of formal verification of these properties on the model. This can be achieved for example by model-checking. The aim of testing w.r.t. security properties is to validate that the properties also hold on the implementation under test (SUT). A formal verification of the SUT is usually out of reach due to its impracticable size.

Now that we are in the context of life-long evolving systems, change has to be taken into account to see the impact it has on a test suite, dedicated to security. We present here an approach from Fraser, Aichernig and Wotawa [36], to handle model changes for regression testing purposes, or to update a test suite. The approach is based on model-checking. It aims at reducing the effort of recreating test suites after a model is changed. It also allows for minimizing the number of regression tests after a change. The considered models are Kripke Structures, i.e. state/transition models. States are labelled with a set of atomic propositions (on the state variables) that hold in this state. A transition relation models the passing from a state to another. A test case is a finite prefix of a path of Kripke structure, with its oracle. It can automatically be converted into a verifiable model [5]. A test suite is a set of test cases, issued from a version of the model.

In case of a model evolution, some of test cases in the test suite issued from the previous version of the model become invalid (i.e. obsolete), while others remain valid. Invalidity is pronounced if the test case goes through a state or a transition that no longer exists in the new model, or if it goes through a state whose labelling has changed.

Ideas presented in the paper permit to decide by model-checking if a test case are still valid after a model change. After what valid test cases can be used as regression tests, whereas the invalid ones can be used as non stagnation tests (to test that what was supposed to change has indeed changed). Additionally, new test cases are created by either adapting the old (invalid) ones (i.e. by re-computing their oracle), or by selectively creating new ones. Model-checking is used to compute or adapt new test cases.

For the creation of the new tests, 3 methods are proposed in [36].

Adaptation. This first method adapts the old test to the new model, by re-computing the oracle of tests from the new model. The test-case model contains a state counter State, and a maximum value MAX. The adaptation can be obtained by querying the model-checker with a particular property, which achieves a trace where the value of State is increased up to MAX. The drawback of this method is that some new behaviours will not be covered, if there are no related obsolete test cases.

Update. The update method is based on trap properties [40]. It is a generalisation of [87] on trapping differences between two versions of a model, by means of a comparator. A trap property is a temporal property dedicated to achieving a given coverage. For example, claiming that a particular (reachable) state cannot be reached achieves the coverage of that state. Depending on which coverage criteria are targeted, a set of trap properties is chosen accordingly. Here, a set P of trap properties is computed for the model before change, and a set P', achieving the same coverage, is computed for the model after change. The new tests are obtained by model-checking trap properties in the difference P $\neq$ P'.

Focus on Model Changes. This method proceeds by automatically rewriting the property and the model before the model-checker is called. The principle of the rewriting is as follows: Rewriting of the model: a boolean variable named change is added to the model (in fact, one boolean variable change is added per change). The change variable is initialized to false, and it takes the true value when the change occurs. It keeps the true value afterwards. Rewriting of the property: all temporal operators in the formula are re-written to include an implication on the change variable. This achieves that only such counter examples are created that include the changed transition.

Attack approach

Usually models that are used for the test generation are supposed to be correct and attack-resistant. In an attack-driven approach, a common practice is to downgrade the model so that he might actually contain an error that can be revealed by an attack. Figure 2.3 illustrates this principle.

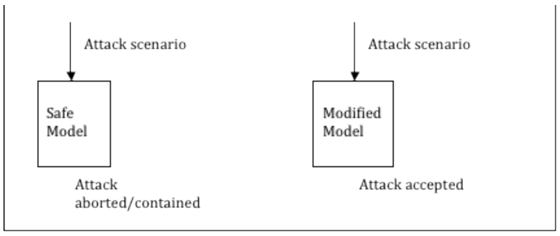


Figure 2.3: Model modification to accept attacks

The model modification makes it possible to play attack scenarios on the model so that the response provides an observable answer to the attack. Once a test representing the attack is actually played on the SUT, if this latter reacts as the modified model, then the SUT presents a weakness.

This approach is called mutation-based testing [24]. Mutations that are introduced are done according to a fault model. It may represent simple syntactical modifications (e.g. replacement of mathematical operators) or more complex ones, motivated by the semantics of the considered model/system.

The modified model is then used in a standard test generation process. Nevertheless, the mutation guides the test generation so as to be able, at test case generation time, to focus on introduced errors [38]. [37] present notions of relevance of test cases w.r.t. (possibly security) properties, based on their error detection capabilities.

In [79], authors present a set of security mutations in access control policies express in OrBAC, it can be used to drive the test generation. In this context, generated tests will be dedicated to requesting, in specific configurations, access to secure data that should be denied by a safe system. The success of the access, and thus, the revealing of the secret, makes it possible to conclude on the presence on errors in the SUT.

In terms of mutation testing for system security robustness, a preliminary work has been done by mutating the model as to simulate environment perturbations that the system has to respond [25]. This kind of mutation can be classified as invasive because it involves non-functional aspects of the SUT.

A related work has be done by [59], based on fault-injection techniques. The idea is to introduce security errors in UMLsec models [58], and to use the UMLsec analysis tools (model-checkers, etc.) to build traces leading to the error. These traces are then used as test cases that are concretized to be executed on the system. The mutations performed are based on adding vulnerabilities in the model, such as missing plausible checks or wrong use of identities, originating from [8].

Recently, an overview of possible vulnerability leaks that may appear in systems, including buffer overflows, SQL injection techniques, etc. This team, in the context of the European project SHIELDS (FP7/2007-203), has proposed a model of vulnerabilities causes, named Vulnerability Cause Graph (VCG), from which is derived a formalism called Vulnerability Detection Condition (VDC) aiming to automatically test the source code to detect vulnerabilities. This test generation is done by the TestIng [18] tool.

3. WP7 Background

In the context of MBT functional and security testing, WP7 partners (INRIA, SMA, UIB) have developed techniques that are considered as the background technology for the Secure Change project. The following sections will provide details on BZTT, Smartesting Test Designer and Telling TestStories.

3.1 Test generation using symbolic animation (BZTT)

We present in this section the BZ-Testing-Tools approach (BZTT) [4, 14], industrialized by the Smartesting company as Test Designer.

3.1.1 Symbolic Animation of B Models

For the test generation approaches to be relevant, it is mandatory to ensure that the model behaves as expected, since the system will be checked against the model. Model animation is thus used for ensuring that the model behaves as described in the initial requirements. This step is done in a semi-automated way, by using a dedicated tool –a model animator– with which the validation engineer interacts. Concretely, the user chooses which operation he wants to invoke. Depending on the current state of the system and the values of the parameters, the animator computes and displays the resulting states that can be obtained. By comparing these states with the informal specification, the user can evaluate its model and correct it if necessary. This process is complementary to the verification that involves properties that have to be formally verified on the model.

The symbolic animation improves the “classical” model animation by giving the possibility to abstract the operation parameters. Once a parameter is abstracted, it is replaced by a symbolic variable that is handled by dedicated constraints solvers. Abstracting all the parameter values turns out to consider each operation as a set of “behaviors", that are the basis from which symbolic animation can be performed [14].

Definition of the Behaviors

A *behavior* is a part of an operation that represents one possible way of executing the operation, in terms of resulting activated effect. Each behavior can be defined as a predicate, representing its activation condition, and a substitution that represents its effect, namely the evolution of the state variables and the instantiation of the return parameters of the operation. The behaviors are computed as the paths in the control flow graph of the considered B operation, represented as a before-after predicate[1].

[1] A before-after predicate is a predicate involving state variables before the operation and after, using a primed notation.

```
sw ← VERIFY_PIN(p) ≙
   PRE p ∈ 0..9999 THEN
     IF tries > 0 ∧ pin ≠ -1 THEN
        IF p = pin THEN
           auth := true ||
           tries := max_tries ||
           sw := ok
        ELSE
           tries := tries - 1 ||
           auth := false ||
           IF tries = 1 THEN
              sw := blocked
           ELSE
              sw := wrong_pin
           END
        END
     ELSE
        sw := wrong_mode
     END
   END
```

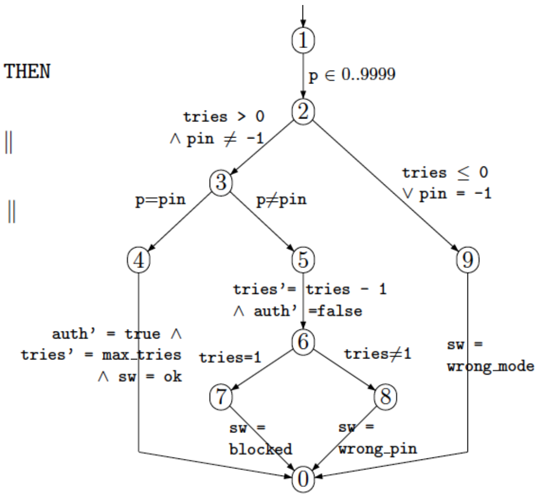


Figure 3.1: B code and control-flow graph of the VERIFY_PIN command

Example 1 (Computation of behaviors) *Consider a smart card command, named VERIFY_PIN aiming at checking a PIN code proposed in parameter against the PIN code of the card. As for every smart card commands, this command returns a code, named* `sw` *for status word, that indicates whether the operation succeeded or not, and possibly indicating the cause of the failure. The precondition specifies the typing information on the parameter* `p` *(a four digit number). First, the command can not succeed if there are no remaining tries on the card and if the current PIN code of the card has been previously set. If the PIN codes match, the card holder is authentified, otherwise there are two cases: either there are enough tries on the card, and the returned status word indicates that the PIN is wrong, or the holder has performed his last try, and the status word indicates that the card is now blocked. This operation is given in Figure 3.1, along with its control flow graph representation. This command presents four behaviors, that are made of the conjunction of the predicates on the edges of a given path, that is denoted by the sequence of nodes from 1 to 0. For example, behavior [1,2,3,4,0], defined by predicate* $\texttt{p} \in 0..9999 \wedge \texttt{tries} > 0 \wedge \texttt{pin} \neq -1 \wedge \texttt{p} = \texttt{pin} \wedge \texttt{auth}' = \texttt{true} \wedge \texttt{tries}' = \texttt{max\_tries} \wedge \texttt{sw} = \texttt{ok}$ *represents a successful authentication of the card holder. In this predicate,* X' *designates the value of variable* X *after the execution of the operation.*

Use of the Behaviors for the Symbolic Animation

When performing the symbolic animation of a B model, the operation parameters are abstracted and thus, the operations are considered through their behaviors. Each parameter is thus replaced by a symbolic variable whose value is managed by a constraint solver.

Definition 2 (Constraint Satisfaction Problem (CSP)) *A Constraint Satisfaction Problem is a triplet* $\langle X, D, C\rangle$ *in which*
- $X = \{X_1, \dots, X_N\}$ *is a set of* N *variables,*
- $D = \{D_1, \dots, D_N\}$ *is a set of domains associated to each variable* $(X_i \in D_i)$*,*
- C *is a set of constraints that relate variable values altogether*
A CSP is said to be consistent *if there exists at least one valuation of the variables in* X *that satisfies the constraints of* C*. It is* inconsistent *otherwise.*

Activating a transition from a given state is equivalent to solving a CSP whose variables X are given by the state variables of the current state (i.e., the state from which the transition is activated), the state variables of after state (i.e., the state reached by the activation of the transition) and the parameters of the operation. Accordingly to the B semantics, the domain D of the variables can be found in the invariant of the machine (resp. in the pre-condition of the operation) for the state variables (resp. for the operation parameters). The contraints C are the predicates composing the behavior that is being activated, enriched with equalities between the before and after variables that are not assigned within the considered behavior.

The feasibility of a transition is defined by the consistency of the CSP associated to the activation of the transition from a given state. The iteration over the possible activable behaviors is done by performing a depth-first exploration of the behavior graph.

Example 2 (Behavior activation) *Consider the activation of the VERIFY_PIN operation given in Example 1. Suppose the activation of this operation from the state s_1 defined by:* `tries = 2`, `auth = false`, `pin = 1234`. *Two behaviors can be activated. The first one corresponds to an invocation* `ok ← VERIFY_PIN(1234)` *that covers path [1,2,3,4,0], and produces the following consistent CSP (notice that data domains have been reduced so as to give the most human-readable representation of the corresponding states):*

$$\begin{aligned} CSP_1 \;=\; & \langle \{tries, auth, pin, p, tries', auth', pin', sw\}, \\ & \{\{2\},\{false\},\{1234\},\{1234\},\{3\},\{true\},\{1234\},\{ok\}\}, \\ & \{Inv, Inv', tries > 0, pin \neq -1, p = pin, tries' = 3, \\ & \qquad auth' = true, pin' = pin, sw = ok\}\rangle \end{aligned} \tag{3.1}$$

where Inv (resp. Inv′) designates the constraints from the machine invariant that apply on the variables before (resp. after) the activation of the behavior. The second activable behavior corresponds to an invocation `wrong_pin ← VERIFY_PIN(`p`)`*, that covers path [1,2,3,5,6,8,0] and produces the following consistent CSP:*

$$\begin{aligned} CSP_2 \;=\; & \langle \{tries, auth, pin, p, tries', auth', pin', sw\}, \\ & \{\{2\},\{false\},\{1234\},0..1233 \cup 1235..9999,\{1\},\{false\},\{1234\},\{wrong\_pin\}\}, \\ & \{Inv, Inv', tries > 0, pin \neq -1, p \neq pin, tries' = tries - 1, \\ & \qquad auth' = false, tries \neq 1, pin' = pin, sw = wrong\_pin\}\rangle \end{aligned} \tag{3.2}$$

State variables may also become symbolic variables, if their after value is related to the value of a symbolic parameter. A variable is said to be symbolic if the domain of the variable contains more than one value. A system state that contains at least one symbolic state variable is said to be a *symbolic state* (by opposition to a concrete state).

Example 3 (Computation of Symbolic States) *Consider the* `SET_PIN` *operation that sets the value of the PIN on a smart card:*

```
sw ← SET_PIN(p) ≙
  PRE p ∈ 0..9999 THEN
     IF pin = -1 THEN  pin := p || sw := ok
     ELSE sw := wrong_mode
     END
  END
```

From the initial state, in which `auth = false`, `tries = 3` *and* `pin = -1`*, the SET_PIN operation can be activated to produce a symbolic state associated to the following CSP:*

$$
\begin{aligned}
CSP_0 \;=\; & \langle \{tries, auth, pin, p, tries', auth', pin', sw\}, \\
& \{\{3\}, \{false\}, \{-1\}, 0..9999, \{3\}, \{false\}, 0..9999, \{ok\}\}, \\
& \{Inv, Inv', pin = -1, pin' = p, sw = ok\}\rangle
\end{aligned} \tag{3.3}
$$

The symbolic animation process works by exploring the successive behaviors of the considered operations. When two operations have to be chained, this process acts as an exploration of the possible combinations of successive behaviors for each operation.

In practice, the selection of the behaviors to be activated is done in a transparent manner and the enumeration of the possible combinations of behaviors chaining is explored using backtracking mechanisms. For animating B models, we use CLPS-BZ [13], a set-theoretical constraint solver written in SICStus Prolog [78] that is able to handle a large subset of the data structures existing in the B machines (sets, relations, functions, integers, atoms, etc.).

Once the sequence has been played, the remaining symbolic parameters can be instantiated by a simple labeling procedure, that consists in solving the constraints system and producing an instantiation of the symbolic variables, obtaining an abstract test case.

It is important to notice that constraint solvers work with an internal representation of constraints (involving constraint graphs and/or polyhedra calculi for relating variable values altogether). Nevertheless, consistency algorithms used to acquire and propagate constraints are not sufficient to ensure the consistency of a set of constraints, and a labelling procedure always has to be employed to guarantee the existence of solutions in a CSP associated to a symbolic state.

The use of symbolic techniques avoids the complete enumeration of the concrete states when animating the model. It thus makes it possible to deal with large models, that represent billions of concrete states, by gathering them into symbolic states.

We now describe the use of symbolic animation for the generation of test cases.

3.1.2 Test Generation

We present in this section the use of the symbolic animation for automating the generation of model-based test cases. This technique aims at a structural coverage of the transitions of the system. To make it simple, each behavior of each operation of the B machine is targeted; the test cases thus aim at covering all the behaviors. In addition, a symbolic representation of the system states makes it possible to perform a boundary analysis from which the test targets will result [64, 3]. This technique is recognized as a pertinent heuristics for generating test data [9].

The tests that we propose are made of four parts, as illustrated in Figure 3.2. The first part, called *preamble*, is a sequence of operations that brings the system from the initial state to a state in which the test target, namely a state from which the considered behavior can be activated, is reached. The *body* is the activation of the behavior itself. Then, the *identification* phase is made of user-defined calls to observation operations, that are supposed to retrieve internal values of the system so that they can be compared to model data in order to establish

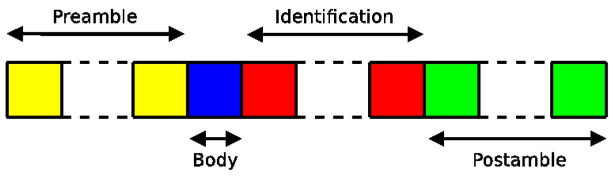


Figure 3.2: Composition of a Test Case

the conformance verdict of the test. Finally, the *postamble* phase is similar to the preamble, but it brings the system back to the initial state or to another state that reaches another test target. The latter part is important to chain the test cases. It is especially used when testing embedded systems, since the execution of the tests on the system is very costly and such systems take usually much time to be reseted by hand.

This automated test generation technique requires some testability hypotheses to be employed. First, the operations of the B machine have to represent the control points of the system to be tested, so as to ease the concretization of the test cases. Second, it is mandatory that the concrete data of the SUT can be compared to the abstract data of the model, so as to be able to compare the results produced by the execution of the test cases with the results predicted by the model. Third, the SUT has to provide observation points that can be modeled in the B machine (either by return values of operations, such as the status words in the smart cards, or by observation operations).

We will now describe how the test cases can be automatically computed, namely how the test targets are extracted from the B machine, and how the test preambles, and postambles, are computed.

Extraction of the Test Targets

The goal of the tests is to verify that the behaviors described in the model exist in the SUT and produce the same result. To achieve that, each test will focus on one specific behavior of an operation. Test targets are defined as the states from which a given behavior can be activated. These test targets are computed so as to satisfy a structural coverage of the machine operations.

Definition 3 (Test Target) *Let $OP = \langle (Act_1, Eff_1)[] \dots [](Act_N, Eff_N)\rangle$ be the set of behaviors extracted from operation OP, in which Act_i denotes the activation condition of behavior i, Eff_i denotes its effect, and $[]$ is an operator of choice between behaviors. Let Inv be the machine invariant. A test target is defined by a predicate that characterizes the states of the invariant from which a behavior i can be activated: $Inv \wedge Act_i$.*

The use of underlying constraint solving techniques makes it possible to provide interesting possibilities for data coverage criteria. In particular, we are able to perform a boundary analysis of the behaviors of the model. Concretely, we will consider *boundary goals*, that are states of the model for which at least one of the state variable is at an extremum (minimum or maximum) of its current domain.

Definition 4 (Boundary Goal) *Let $minimize(V, C)$ (resp. $maximize(V, C)$) be a function that instantiates a symbolic variable V to its minimal value (resp. its maximal value), under the constraints given in C. Let Act_i be the activation condition of behavior i, let $\vec{P}$ be the parameters of the corresponding operation, and let $\vec{V}$ be the set of state variables that occur in behavior i, the boundary goals for the variables $\vec{V}$ are computed by:*

$$BG^{min} = minimize(f(\vec{V}), Inv \wedge \exists \vec{P}.Act_i)$$

$$BG^{max} = maximize(f(\vec{V}), Inv \wedge \exists \vec{P}.Act_i)$$

in which f is an optimization function that depends on the type of the variable:

if $\vec{X}$ is a set of integers, $f(\vec{X}) = \sum_{x \in \vec{X}} x$

if $\vec{X}$ is a set of sets, $f(\vec{X}) = \sum_{x \in \vec{X}} card(x)$

otherwise, $f(\vec{X}) = 1$

N°	Rewriting of $P_1 \vee P_2$	Coverage criterion
1	$P_1 \vee P_2$	Decision Coverage (DC)
2	$P_1 \ [] \ P_2$	Condition/Decision Coverage (C/DC)
3	$P_1 \wedge \neg P_2 \ [] \ \neg P_1 \wedge P_2$	Full Predicate Coverage (FPC)
4	$P_1 \wedge P_2 \ [] \ P_1 \wedge \neg P_2 \ [] \ \neg P_1 \wedge P_2$	Multiple Condition Coverage (MCC)

Table 3.1: Decision coverage criteria depending on rewritings

Example 4 (Boundary test targets) *Consider behavior [1,2,3,4,5,0] from the* `VERIFY_PIN` *operation presented in Figure 3.1. The machine invariant gives the following typing informations:*

$$Inv \mathrel{\hat{=}} tries \in 0..3 \wedge pin \in -1..9999 \wedge auth \in \{true, false\}$$

The boundary test targets are computed using the minimization/maximization formulas:

$$\begin{aligned} BG^{min} &= minimize(tries + pin, Inv \wedge \exists p \in 0..9999.(tries > 0 \wedge pin \neq -1 \wedge pin = p)) \\ &\quad \rightsquigarrow tries = 1,\ pin = 0 \\ BG^{max} &= maximize(tries + pin, Inv \wedge \exists p \in 0..9999.(tries > 0 \wedge pin \neq -1 \wedge pin = p)) \\ &\quad \rightsquigarrow tries = 3,\ pin = 9999 \end{aligned}$$

In order to improve the coverage of the operations, a predicate coverage criterion [67] can be applied by the validation engineer. This criterion acts as a rewriting of the disjunctions in the decisions of the B machine. Four rewritings are possible, that make it possible to satisfy different specification coverage criteria [67], as given in Table 3.1.

Rewriting 1 leaves the disjunction unmodified. Thus, the Decision Coverage criterion will be satisfied if a test target satisfies either P_1 or P_2 indifferently (also satisfying the Condition Coverage (CC) criterion). Rewriting 2 produces two test targets, one considering the satisfaction of P_1, the other the satisfaction of P_2. Rewriting 3 will also produce two test targets, considering an exclusive satisfaction of P_1 without P_2 and vice-versa. Finally, Rewriting 4 produces three test targets that will cover all the possibilities to satisfy the disjunctions.

Notice that the consistency of the resulting test targets is checked so as to eliminate inconsistent test targets.

Example 5 (Decision coverage) *Consider behavior [1,2,9,0] from operation* `VERIFY_PIN` *presented in Figure 3.1. The selection of the Multiple Condition Coverage criterion will produce the following test targets:*

1. $Inv \wedge \exists p \in 0..9999 \ . \ (tries \leq 0 \wedge pin = -1)$
2. $Inv \wedge \exists p \in 0..9999 \ . \ (tries > 0 \wedge pin = -1)$
3. $Inv \wedge \exists p \in 0..9999 \ . \ (tries \leq 0 \wedge pin \neq -1)$

providing contexts from which boundary goals will then be computed.

We now describe how these targets are reached by symbolic animation by computation of the test preamble.

Computation of the Test Cases

Once the test targets and boundary goals are defined, the idea is to employ symbolic animation in an automated manner that will aim at reaching each target. To achieve that, a state

```
SeqOp ← compute_preamble(Depth, Target)
begin
   $s\_init$ ← initialize ;
   $Seq\_curr$ ← [init] ;
   $dist\_init$ ← compute_distance(Target,$s\_init$) ;
   visited ← [⟨ $s\_init, Seq\_curr, dist\_init$ ⟩] ;
    while visited $\neq$ [] do
      ⟨ $s\_curr$, $Seq\_curr$, $MinDist$ ⟩ ← remove_minimal_distance(visited) ;
      if length($Seq\_curr$) < Depth then
         [$(s\_1, Seq\_1), \ldots, (s\_N, Seq\_N)$] ← compute_successors(($s\_curr, Seq\_curr$)) ;
         for each ($s\_i, Seq\_i$) $\in$ [$(s\_1, Seq\_1), \ldots, (s\_N, Seq\_N)$] do
            $dist\_i$ ← compute_distance(Target,$s\_i$) ;
            if $dist\_i$ = 0 then
               return $Seq\_i$;
            else
               visited ← visited $\cup$ ($s\_i$, $Seq\_i$, $dist\_i$) ;
            end if
         done
      end if
   done
   return [];
end
```

Figure 3.3: State exploration algorithm

exploration algorithm, variant of the A* path-finding algorithm and based on a Best-First exploration of the system states, has been developed.

This algorithm aims at finding automatically a path, from the initial state, that will reach a given set of states characterized by a predicate. A sketch of the algorithm is given in Figure 3.3. From a given state, the symbolic successors, through each behavior, are computed using symbolic animation (procedure `compute_successors`). Each of these successors is then evaluated to compute the distance to the target. This latter is based on a heuristics that considers the "distance" between the current state and the targeted states (procedure `compute_distance`). To do that, the sum of the distances between each state variable is considered; if the domains of the two variables intersect, then the distance for these variables is 0, otherwise a customized formula, involving the type of the variable and the size of the domains, computes the distance (see [21] for more details). The computation of the sequence restarts from the most relevant state, i.e., the one presenting the smallest distance to the target (procedure `remove_minimal_distance` returning the most interesting triplet ⟨state, sequence of behaviors, distance⟩ and removing it from the list of visited states). The algorithm starts with the initial state (denoted by s_init and obtained by initializing the variables according to the INITIALIZATION clause of the machine denoted by the `initialize` function). It ends if a zero-distance state is reached by the current sequence, or if all sequences have been explored for a given depth.

Since reachability of the test targets can not be decided, this algorithm is bounded in depth. Its worst case complexity is $O(n^d)$ where n is the number of behaviors in all the operations of the machine and d is the depth of the exploration (maximal length of test sequence). Nevertheless, the heuristics consisting in computing the distance between the states explored and the targeted states to select the most relevant states improves the practical results of the algorithm.

The computation of the preambule ends for three possible reasons. It may have found the target, and thus, the path is returned as a sequence of behaviors. Notice that, in practice, this path is often the shortest from the initial state, but it is not always the case because of the heuristics used in during the search. The algorithm may also end by stating that the target has not been reached. This can be due to the fact that the exploration depth was too small, but it may also be due to the unreachability of the target.

Example 6 (Reachability of the test targets) *Consider the three targets given in Example 5. The last two can easily be reached. Target 2 can be reached by setting the value of the PIN, and Target 3 can be reached by setting the value of the PIN, followed by three successive authentication failures.*

Nevertheless, the first target will never be reached since the decrementation of the tries can only be done if pin ≠ -1. In order to avoid considering unreachable targets, the machine invariant has to be complete enough to catch at best the reachable states of the system, or, at least, to exclude unreachable states. In the example, completing the invariant by: $pin = -1 \Rightarrow tries = 3$ *makes Target 1 inconsistent, and thus, removes it from the test generation process.*

The sequence returned by the algorithm represents the preamble, to which is concatenated the invocation of the considered behavior (representing the test body). If operation parameters are still constrained, they are also minimized or maximized, for their instantiation. The observation operations are specified by hand, and the (optional) postamble is computed on the same principle as the preamble.

3.2 Test Designer (TD)

Test Designer, from Smartesting, is a commercially available model-based testing tool dedicated to IT applications, secure electronic transactions and packaged applications such as SAP or Oracle E-Business Suite.

Test Designer implements concepts presented in the previous section based on a before/after semantics, symbolic animation and preamble computation on a subset of UML/OCL notation [15].
Test cases are generated from a behavior model of the SUT, using requirement coverage and custom scenarios as test selection criteria. Test Designer models are written in a subset of standard UML (class and object diagrams as well as state machine diagrams, with OCL annotations). Test Designer supports both a transition-based modeling style (i.e. UML State Machines) and a Pre/Post style (i.e. OCL).
Model elements such as transitions and OCL decisions can be linked to the informal requirements that they cover. Test coverage can then be based on requirements coverage. A range of structural model coverage criteria are also supported, such as transition, decision and effect coverage. The test engineer may also define business scenarios as custom test case specifications that use the UML operations. Test Designer supports both manual and automated test execution, using an offline approach. The generated test cases can be output to test management systems like HP Quality Center or IBM Rational Quality Manager, with bidirectional traceability and full change management for evolving requirements.

Now, we illustrate how Test Designer can be used to model and test the actiTIME application (www.actitime.com).

actiTIME is a time tracking application freely available on the web. In this section, we use this application to demonstrate the various steps of deploying the MBT process. We illustrate it with Test Designer from Smartesting, which is a model-based testing solution dedicated to enterprise IT applications, secure electronic transactions and packaged applications such as SAP or Oracle E-Business Suite. Test cases are generated from a behavior model of the SUT, using requirements coverage and custom scenarios as test selection criteria. Test Designer models are written in a subset of standard UML.
Test Designer supports both manual and automated test execution, using an offline approach. The generated test cases can be output to test management systems like HP Quality Center, IBM Rational Quality Manager or the open-source tool TestLink, with bidirectional traceability and full change management for evolving requirements.

3.2.1 actiTIME overview

actiTIME is a time management program developed by Actimind. Details about its features, and free downloads, can be found on the website www.actitime.com. Ordinary users have access to their time track for input, review and corrections. They can also manage projects and tasks, do some reporting and of course they can manage their account (see (1) in Figure 3.4).
In our sample model we focus on the user time-tracking features of actiTIME version 1.5; after logging into the system the user can specify how much time he spent on a specific task. A typical scenario is as follows:

1. access a time-track;
2. display the time-entry form;
3. type in the hours spent on assigned tasks;
4. the system warns the user that modifications are not saved yet;
5. save the modifications;
6. in case of overtime, the system displays an error message.

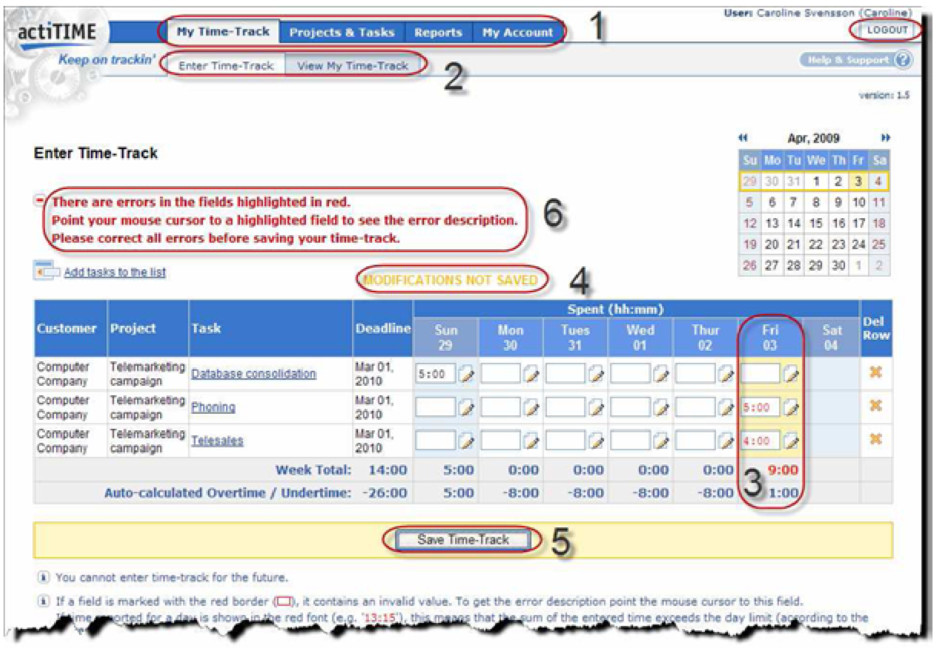
3.2.2 actiTIME requirements

In actiTIME a user may have administrator rights. Only administrators can add and remove projects. For a specific project, a user can add or remove tasks, enter the number of hours they spent on a task, etc. To precisely define the expected functional requirements of the actiTIME feature that we model, a list of requirements is defined in Table 3.2. The IDs are useful as they allow you to see in the test repository which requirements are covered by each of the generated tests.

3.2.3 actiTIME test model

The test model represents the expected behavior of the application, covering the requirements of Table 3.2. It is based on three UML diagrams (see Figure 3.5, Figure 3.6 and Figure 3.7):

- the class diagram represents the business entities and the user actions to be tested;



Enter Time-Track

There are errors in the fields highlighted in red.
Point your mouse cursor to a highlighted field to see the error description.
Please correct all errors before saving your time-track.

Apr, 2009						
Su	Mo	Tu	We	Th	Fr	Sa
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2

Customer	Project	Task	Deadline	Spent (hh:mm)							Del Row
				Sun 29	Mon 30	Tues 31	Wed 01	Thur 02	Fri 03	Sat 04	
Computer Company	Telemarketing campaign	Database consolidation	Mar 01, 2010	5:00							✖
Computer Company	Telemarketing campaign	Phoning	Mar 01, 2010						5:00		✖
Computer Company	Telemarketing campaign	Telesales	Mar 01, 2010						4:00		✖
		Week Total:	14:00	5:00	0:00	0:00	0:00	0:00	9:00		
		Auto-calculated Overtime / Undertime:	-26:00	5:00	-8:00	-8:00	-8:00	-8:00	1:00		

You cannot enter time-track for the future.

If a field is marked with the red border (▭), it contains an invalid value. To get the error description point the mouse cursor to this field.

If time reported for a day is shown in the red font (e.g. '13:15'), this means that the sum of the entered time exceeds the day limit (according to the

Figure 3.4: actiTIME user interface

Requirement Id	Requirement Description
ADMIN/ADD_PROJECT	An administrator can add a new project into the system. A project is linked to a customer and includes several tasks.
ADMIN/DELETE_PROJECT	An administrator can delete a project from the system.
LOGIN	When the user try to log on with an incorrect username or password an error message is displayed.
USER/ENTER_TIME	A user can enter the number of hours spent on his assigned tasks for one or several days.
USER/REMOVE_TIME	A user can correct the number of hours spent on a task by removing some time.
USER/SAVE_TIME	After modifying its time-track, the user can save the changes.
USER/SHOW_TIME_TRACKING	A user can display its time-track consolidation for any month.
USER/VIEW_TIME_TRACK	A user can display its time-track for the current week or any week, in order to report its activity.
USER/WORKING_TASK	A user can add or remove task from the task list.

Table 3.2: Summary of actiTIME requirements.

- the layered state machine represents the dynamic expected behavior;
- the instance diagram gives some test data and initial configuration of the application.

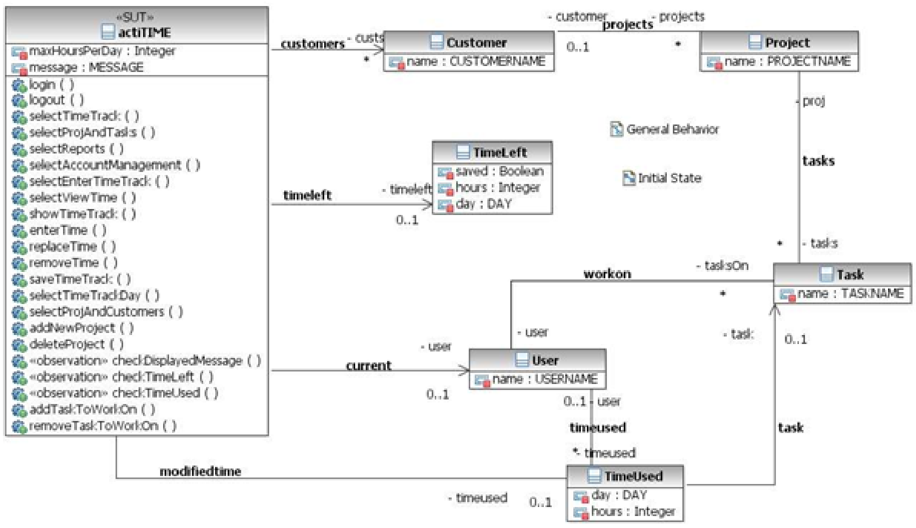


Figure 3.5: Class diagram for actiTIME test model

Figure 3.8 gives an OCL specification describing the Login operation with an invalid user name. Notice how the requirements are linked to the specification using annotations (—@Req: LOGIN). The annotation —@AIM gives more detail about which part of that refinement is being modeled here.

3.2.4 Test generation with Test Designer

Figure 3.9 shows the GUI of Test Designer for the project actiTIME. A list of the generated test cases (structured by test Suites) is displayed on the left, and the details of one test case are displayed on the right. The details of the requirements and test aims that are covered by a particular test step are shown in the right-hand bottom corner.

Figure 3.10 shows the generated tests published into a test repository (in this case: HP Quality Center). These tests are ready for manual test execution. Each test is fully documented in the Design Steps Panel.

For test automation, complete script code is generated and maintained for each test case (see Figure 3.11). The remaining (optional) task for the test automation engineer is to implement each key-word used in UML test model so that it is defined as a sequence of lower-level SUT actions. If this is done, the generated test scripts can be executed automatically on the SUT. An alternative approach is to leave the key-words undefined, in which case a human tester must execute the scripts manually.

To sum-up, we deployed on the actiTIME application a typical MBT solution for IT applications, using a subset of UML as input language (class diagrams, state diagrams, instance diagrams, and OCL specification language), providing automated test generation and publication features both for manual and automated testing.

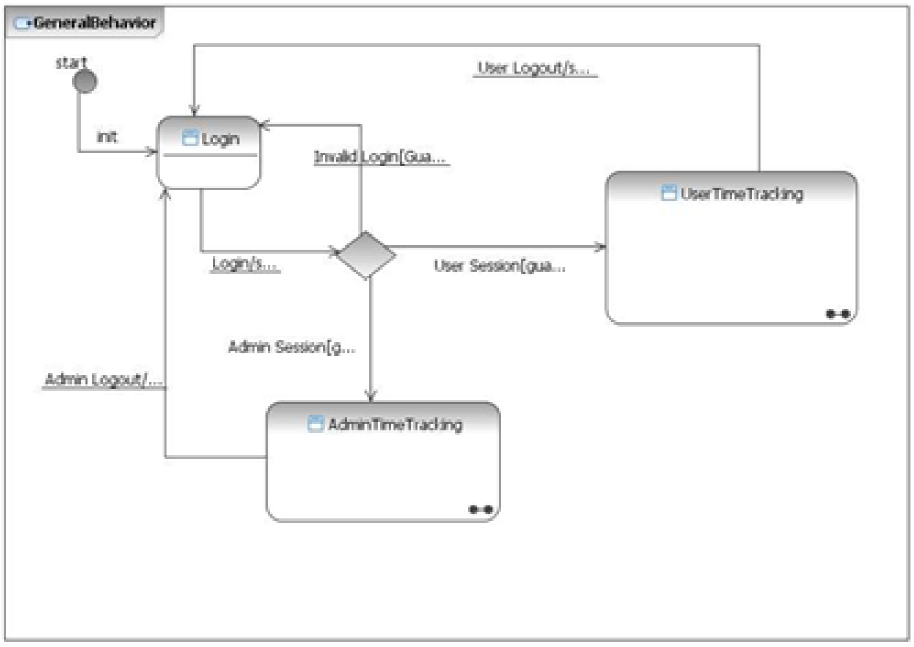


Figure 3.6: High level state machine for actiTIME (partial)

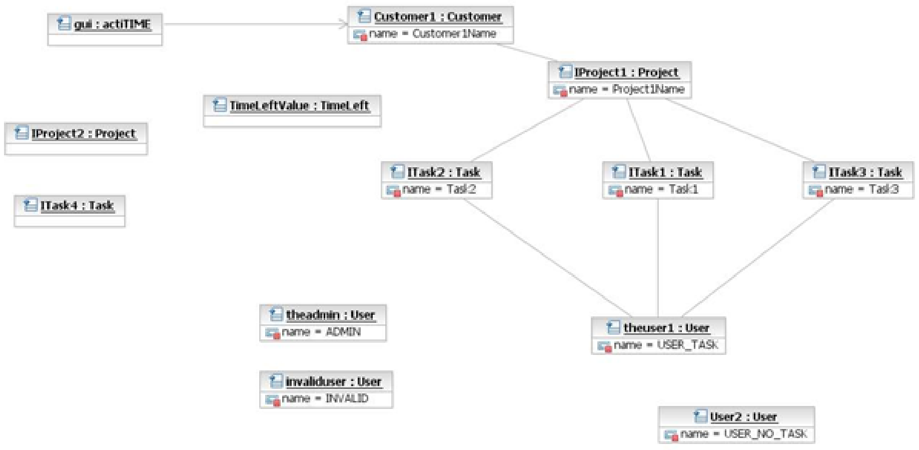
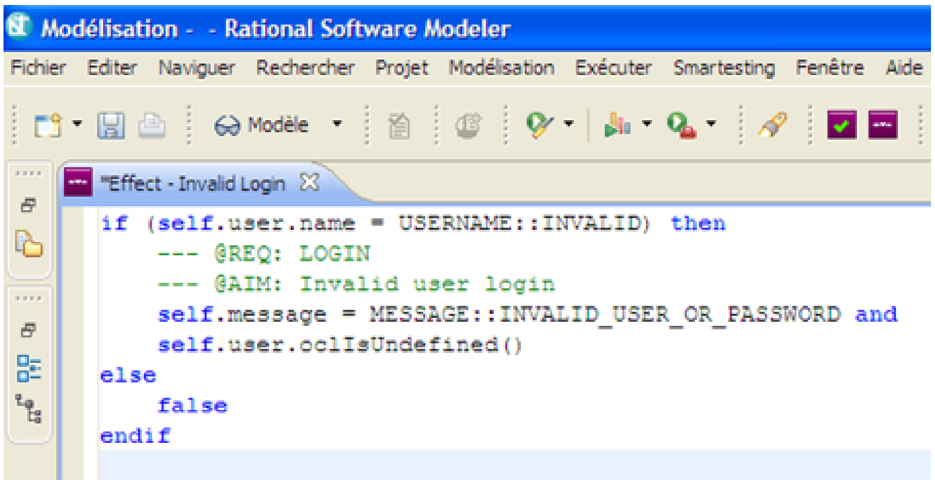
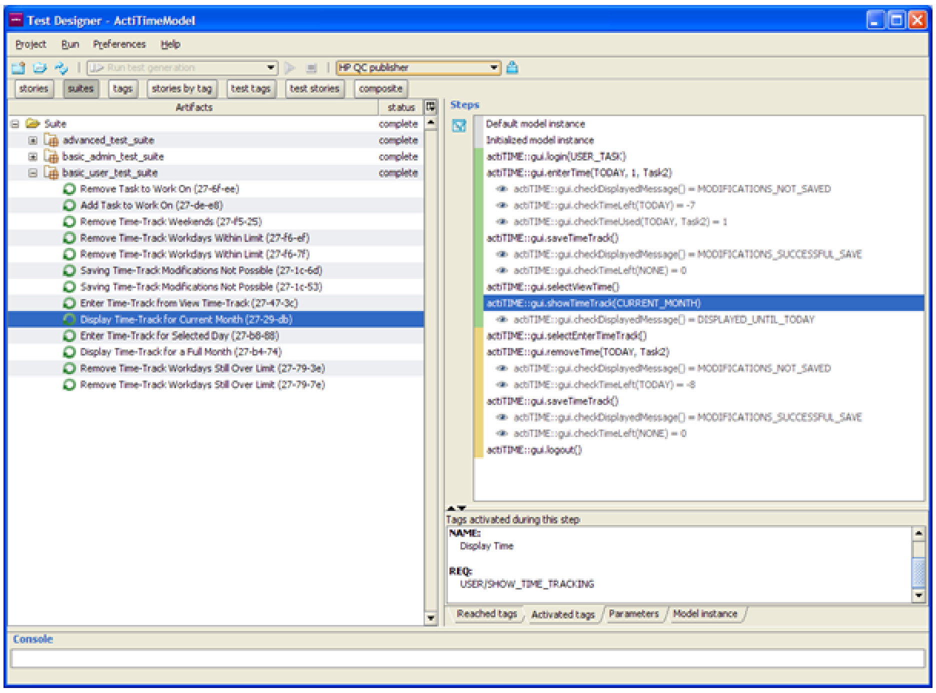


Figure 3.7: Object diagram for actiTIME


Modélisation - - Rational Software Modeler
Fichier Editer Naviguer Rechercher Projet Modélisation Exécuter Smartesting Fenêtre Aide
Modèle
*Effect - Invalid Login

```
if (self.user.name = USERNAME::INVALID) then
    --- @REQ: LOGIN
    --- @AIM: Invalid user login
    self.message = MESSAGE::INVALID_USER_OR_PASSWORD and
    self.user.oclIsUndefined()
else
    false
endif
```

Figure 3.8: OCL specification for Login (the invalid login case).


Test Designer - ActiTimeModel
Project Run Preferences Help
Run test generation
HP QC publisher
stories suites tags stories by tag test tags test stories composite
Steps
Reached tags Activated tags Parameters Model instance
Console

Artifacts	status
Suite	complete
advanced_test_suite	complete
basic_admin_test_suite	complete
basic_user_test_suite	complete
Remove Task to Work On (27-6f-ee)	
Add Task to Work On (27-de-e8)	
Remove Time-Track Weekends (27-f5-25)	
Remove Time-Track Workdays Within Limit (27-f6-ef)	
Remove Time-Track Workdays Within Limit (27-f6-7f)	
Saving Time-Track Modifications Not Possible (27-1c-6d)	
Saving Time-Track Modifications Not Possible (27-1c-53)	
Enter Time-Track from View Time-Track (27-47-3c)	
Display Time-Track for Current Month (27-29-db)	
Enter Time-Track for Selected Day (27-b8-88)	
Display Time-Track for a Full Month (27-b4-74)	
Remove Time-Track Workdays Still Over Limit (27-79-3e)	
Remove Time-Track Workdays Still Over Limit (27-79-7e)	

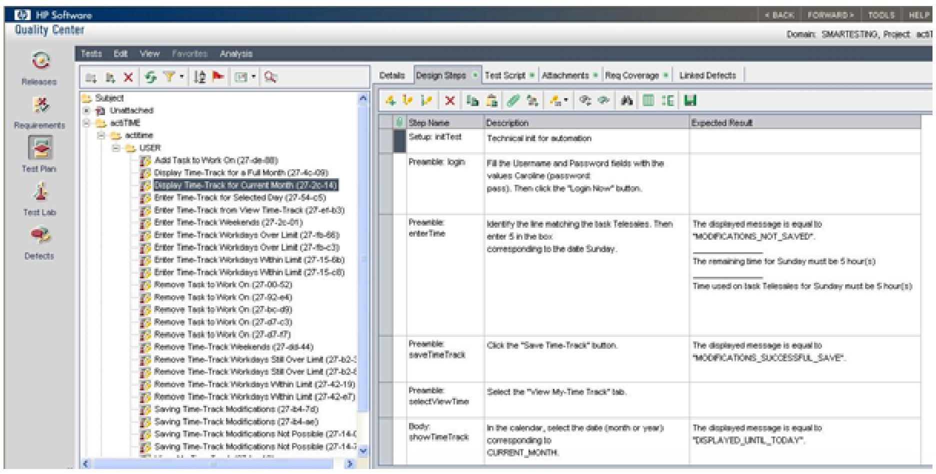
```
Default model instance
Initialized model instance
actiTIME::gui.login(USER_TASK)
actiTIME::gui.enterTime(TODAY, 1, Task2)
    actiTIME::gui.checkDisplayedMessage() = MODIFICATIONS_NOT_SAVED
    actiTIME::gui.checkTimeLeft(TODAY) = -7
    actiTIME::gui.checkTimeUsed(TODAY, Task2) = 1
actiTIME::gui.saveTimeTrack()
    actiTIME::gui.checkDisplayedMessage() = MODIFICATIONS_SUCCESSFUL_SAVE
    actiTIME::gui.checkTimeLeft(NONE) = 0
actiTIME::gui.selectViewTime()
actiTIME::gui.showTimeTrack(CURRENT_MONTH)
    actiTIME::gui.checkDisplayedMessage() = DISPLAYED_UNTIL_TODAY
actiTIME::gui.selectEnterTimeTrack()
actiTIME::gui.removeTime(TODAY, Task2)
    actiTIME::gui.checkDisplayedMessage() = MODIFICATIONS_NOT_SAVED
    actiTIME::gui.checkTimeLeft(TODAY) = -8
actiTIME::gui.saveTimeTrack()
    actiTIME::gui.checkDisplayedMessage() = MODIFICATIONS_SUCCESSFUL_SAVE
    actiTIME::gui.checkTimeLeft(NONE) = 0
actiTIME::gui.logout()
```

Tags activated during this step

NAME:
Display Time

REQ:
USER/SHOW_TIME_TRACKING

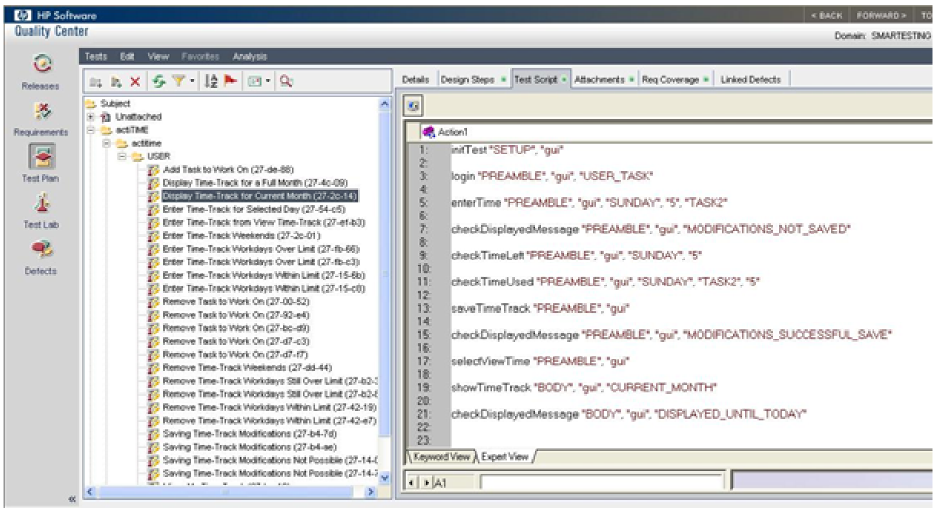
Figure 3.9: Smartesting Test Designer user interface. Project actiTIME.



- Subject
  - Unattached
  - actiTIME
    - actitime
      - USER
        - Add Task to Work On (27-de-88)
        - Display Time-Track for a Full Month (27-4c-09)
        - Display Time-Track for Current Month (27-2c-14)
        - Enter Time-Track for Selected Day (27-54-c5)
        - Enter Time-Track from View Time-Track (27-ef-b3)
        - Enter Time-Track Weekends (27-2c-01)
        - Enter Time-Track Workdays Over Limit (27-fb-66)
        - Enter Time-Track Workdays Over Limit (27-fb-c3)
        - Enter Time-Track Workdays Within Limit (27-15-6b)
        - Enter Time-Track Workdays Within Limit (27-15-c8)
        - Remove Task to Work On (27-00-52)
        - Remove Task to Work On (27-92-e4)
        - Remove Task to Work On (27-bc-d9)
        - Remove Task to Work On (27-d7-c3)
        - Remove Task to Work On (27-d7-f7)
        - Remove Time-Track Weekends (27-dd-44)
        - Remove Time-Track Workdays Still Over Limit (27-b2-
        - Remove Time-Track Workdays Still Over Limit (27-b2-
        - Remove Time-Track Workdays Within Limit (27-42-19)
        - Remove Time-Track Workdays Within Limit (27-42-e7)
        - Saving Time-Track Modifications (27-b4-7d)
        - Saving Time-Track Modifications (27-b4-ae)
        - Saving Time-Track Modifications Not Possible (27-14-
        - Saving Time-Track Modifications Not Possible (27-14-

Step Name	Description	Expected Result
Setup: initTest	Technical init for automation	
Preamble: login	Fill the Username and Password fields with the values Caroline (password: pass). Then click the "Login Now" button.	
Preamble: enterTime	Identify the line matching the task Telesales. Then enter 5 in the box corresponding to the date Sunday.	The displayed message is equal to "MODIFICATIONS_NOT_SAVED". The remaining time for Sunday must be 5 hour(s) Time used on task Telesales for Sunday must be 5 hour(s)
Preamble: saveTimeTrack	Click the "Save Time-Track" button.	The displayed message is equal to "MODIFICATIONS_SUCCESSFUL_SAVE".
Preamble: selectViewTime	Select the "View My-Time Track" tab.	
Body: showTimeTrack	In the calendar, select the date (month or year) corresponding to CURRENT_MONTH.	The displayed message is equal to "DISPLAYED_UNTIL_TODAY".

Figure 3.10: Publication of generated tests into the test manager environment (HP Quality Center)



- Subject
  - Unattached
  - actiTIME
    - actitime
      - USER
        - Add Task to Work On (27-de-88)
        - Display Time-Track for a Full Month (27-4c-09)
        - Display Time-Track for Current Month (27-2c-14)
        - Enter Time-Track for Selected Day (27-54-c5)
        - Enter Time-Track from View Time-Track (27-ef-b3)
        - Enter Time-Track Weekends (27-2c-01)
        - Enter Time-Track Workdays Over Limit (27-fb-66)
        - Enter Time-Track Workdays Over Limit (27-fb-c3)
        - Enter Time-Track Workdays Within Limit (27-15-6b)
        - Enter Time-Track Workdays Within Limit (27-15-c8)
        - Remove Task to Work On (27-00-52)
        - Remove Task to Work On (27-92-e4)
        - Remove Task to Work On (27-bc-d9)
        - Remove Task to Work On (27-d7-c3)
        - Remove Task to Work On (27-d7-f7)
        - Remove Time-Track Weekends (27-dd-44)
        - Remove Time-Track Workdays Still Over Limit (27-b2-
        - Remove Time-Track Workdays Still Over Limit (27-b2-
        - Remove Time-Track Workdays Within Limit (27-42-19)
        - Remove Time-Track Workdays Within Limit (27-42-e7)
        - Saving Time-Track Modifications (27-b4-7d)
        - Saving Time-Track Modifications (27-b4-ae)
        - Saving Time-Track Modifications Not Possible (27-14-
        - Saving Time-Track Modifications Not Possible (27-14-

```
initTest "SETUP", "gui"

login "PREAMBLE", "gui", "USER_TASK"

enterTime "PREAMBLE", "gui", "SUNDAY", "5", "TASK2"

checkDisplayedMessage "PREAMBLE", "gui", "MODIFICATIONS_NOT_SAVED"

checkTimeLeft "PREAMBLE", "gui", "SUNDAY", "5"

checkTimeUsed "PREAMBLE", "gui", "SUNDAY", "TASK2", "5"

saveTimeTrack "PREAMBLE", "gui"

checkDisplayedMessage "PREAMBLE", "gui", "MODIFICATIONS_SUCCESSFUL_SAVE"

selectViewTime "PREAMBLE", "gui"

showTimeTrack "BODY", "gui", "CURRENT_MONTH"

checkDisplayedMessage "BODY", "gui", "DISPLAYED_UNTIL_TODAY"

```

Figure 3.11: Publication of generated scripts into the test manager environment (HP Quality Center)

3.3 Telling TestStories

Model-driven testing is based on the derivation of executable test code from models in a MDA manner [89]. It supports an implementation resp. technology-independent view on testing and the adaptation of tests to modified requirements with minor effort. This makes model-driven testing an ideal testing strategy for service oriented systems. In such a setting various component- and communication technologies, the dynamic adaptation and integration of services and the unavailability of service implementations and controls have to be considered.

Telling TestStories (TTS) [30] provides a testing methodology and a framework for model-driven system testing of service oriented systems. Compared to many other model-based testing approaches, TTS is based on separated system and test models which are connected via common model elements. An overview of the TTS artefacts is depicted in Figure 3.12.

The requirements model contains the specification for system development. Its formal part consists of actors, use cases and types, denoted in a use case diagram and a class diagram. The formal requirements are based on written or non-written informal requirements.

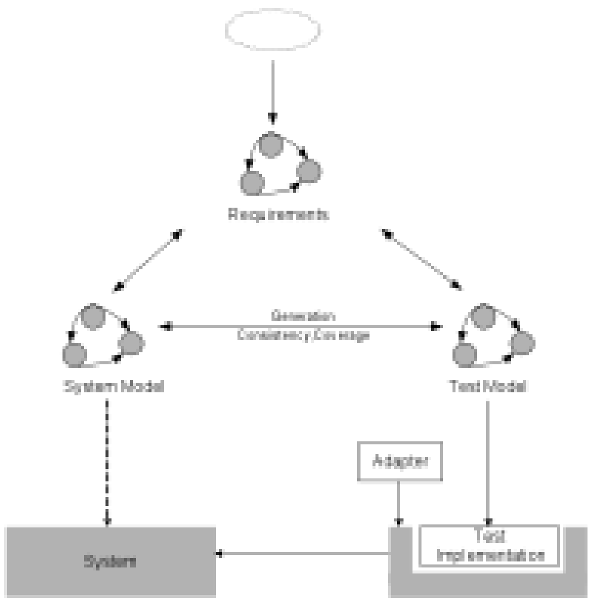


Figure 3.12: Overview of TTS Artefacts

The **system model** describes the system structure and system behavior in a platform independent way. Its static structure is based on actors providing and requiring services and its dynamic structure is based on global workflows modeling the behavior between actors and local workflows modeling the behavior within actors.

The **test model** defines the test configuration, the test data and the test scenarios as so called test stories. Test stories are controlled sequences of service operation invocations exemplifying the interaction of actors. Test stories may be generic in the sense that they do not contain concrete objects but variables which refer to test values provided in tables. Test stories can also contain setup and tear down procedures and contain assertions for test result evaluation. The service operations and actors of a test story are shared with the system model.

The **test implementation** is generated by a model-to-text transformation explained in [31]. It generates test code that can be executed by a test execution engine. Adapters are needed for connecting to the system under test.

Metamodels have been defined to define consistency and coverage criteria for the system and the test model.

If the system model and the test model are created manually, then the framework checks consistency between them automatically and the test model has to fulfill some coverage properties with respect to the system model. Alternatively, if the system model is complete to a certain sense then behavioral parts of the test model can be generated or otherwise if the test model is complete, behavioral fragments of the system model can be generated.

The metamodel elements can be mapped to UML metaclasses and can therefore be created and edited with standard UML tools. The system model can even be mapped to SoaML and the test model can be mapped to the UML Testing Profile which ensures compatibility with existing standards.

A very important property of TTS is the traceability between system requirements, service operations, test stories and the implementation. This enables the assignment of unexpected system behavior to requirements resp. to system or test model elements. Therefore, the system and test model can be created, transformed, executed and analyzed iteratively even in a test-driven manner based on changing requirements and services. Furthermore, TTS allows for the execution of tests during system development by creating mock services for unfinished parts of a system.

4. Bottlenecks addressed by WP7

This chapter proposes an analysis of commercial tools based on the criteria identified in Chapter 1. We also provide several research directions that will be investigated during the project.

4.1 Evaluation of existing approaches

In chapter 1, we proposed several criteria to evaluate MBT methods with respect to change and security aspects. In chapter 2, we introduced a state of the art for functional model-based testing.

So, in this section, we evaluate several MBT tools with respect to these criteria. Those tools all use as input behavioral model of the SUT and provide various test coverage criteria. Based on the taxonomy paper [83] by Utting and al., we have selected a list of commercial tools to be compared presented in Table 4.1 and Table 4.2.

Tools	Stability of test repository	Traceability of changes	Impact analysis	Test suite qualification
Conformiq Qtronic	2	1	1	no qualification
Microsoft SpecExplorer	1	1	1	no qualification
Reactive Systems Reactis	2	1	1	no qualification
Smartesting Test Designer	2	1	1	no qualification
T-Vec Tool Suite	1	1	1	no qualification

Table 4.1: Change criteria applied on commercial tools

Tools	Traceability of security properties	Completeness of security testing
Conformiq Qtronic	2	functional security properties only
Microsoft SpecExplorer	1	functional security properties only
Reactive Systems Reactis	1	functional security properties only
Smartesting Test Designer	2	functional security properties only
T-Vec Tool Suite	2	functional security properties only

Table 4.2: Security criteria applied on commercial tools

To sum-up, MBT methods for testing the security aspects of long-life evolving systems is still an open subject. The issues of the stability of the generated test repository, the impact analysis of change in relation with testing security properties, the traceability between security properties and generated tests in the context of evolution, are not tackled by the current MBT methods.

4.2 Research directions for WP7

For WP7, the next step of the work is to prepare a dedicated approach based on the existing MBT background SecureChange partners presented in this document (see Chapter 4). We will study the evolution's impact on model-based testing approaches with WP1 case studies.

The first research direction is on test status. With respect to the state of the art, we will integrate the concept of test lifecycle to take into account the evolution management meaning that the test repository must evolve with respect with requirements and SUT evolutions. One of the main element of this point is the minimization of evolution's impacts on the test repository. We will define a lifecycle and an associated method to provide a test's classification based on design artefacts to ensure repository stability. Actually, there are no tools and methods that take into account evolution. The only actual answer is to replay a selected test on the test model. This method can only decide if a test should or should not be kept for the following version of the software.

The second direction is the identification of evolution's sources. To capture evolution's information, we will identify each kind of evolutions which impacts the testing process. We use WP7 background to define the input artefacts and the test generation method. There are two input artefacts : the functional model and the associated scenario. Artefacts are produced from requirements that are used at two levels.

1. Test suites are designed based on requirement coverage criteria (e.g. functional requirements, security requirements, etc).
2. The traceability process relates the tests to their corresponding informal requirements. Their evolution can be taken into account for distinguishing which tests (linked to evolved requirements) are supposed to be replaced, and which tests (linked to unchanged requirements) will be used for non-regression testing.

The third direction is the evaluation of the evolution's impacts on security. Scenario will be used to generate test with respect to security properties based on SUT expected behaviour defined into the functional model. After the study of evolution's impacts on the model, we will study their impacts on the scenario.

The last direction will be the method's application on case studies. We will model the software behaviour of two case studies. We will be able to validate our methods and algorithms of classifications on real-life evolving systems.

5. Conclusion

Deliverable 7.1 provides a state of the art of MBT methods and introduces the background technologies that constitute the starting point of the project.

We identified in the state of the art several criteria to evaluate testing approaches. We applied the defined criteria to evaluate model-based testing approaches with respect to evolution and security of systems.

We propose four research directions to take into account evolution for security. Those research directions will be investigated during the project.

Bibliography

[1] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] Bernhard K. Aichernig, Harald Brandl, and Franz Wotawa. Conformance testing of hybrid systems with qualitative reasoning models. *Electron. Notes Theor. Comput. Sci.*, 253(2):53–69, 2009.

[3] F. Ambert, F. Bouquet, B. Legeard, and F. Peureux. Automated boundary-value test generation from specifications - method and tools. In *4th Int. Conf. on Software Testing, ICSTEST 2003*, pages 52–68, Cologne, Allemagne, April 2003.

[4] Fabrice Ambert, Fabrice Bouquet, Sébastien Chemin, Sébastien Guenaud, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proc. of Formal Approaches to Testing of Software, FATES 2002 (workshop of CONCUR'02)*, pages 105–120, Brnö, République Tchèque, August 2002. INRIA report.

[5] Paul Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *HASE*, pages 239–248. IEEE Computer Society, 1999.

[6] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *ICFEM*, pages 46–, 1998.

[7] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *STTT*, 11(1):69–83, 2009.

[8] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. Use of a taxonomy of security faults. In *Purdue University*, pages 551–560, 1996.

[9] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[10] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[11] Ottmar Beucher. *MATLAB und Simulink (Scientific Computing)*. Pearson Studium, 08 2006.

[12] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[13] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157, August 2004.

[14] F. Bouquet, B. Legeard, M. Utting, and N. Vacelet. Faster analysis of formal specifications. In J. Davies, W. Schulte, and M. Barnett, editors, *6th Int. Conf. on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 239–258, Seattle, WA, USA, November 2004. Springer-Verlag.

[15] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 45–48, Leipzig, Germany, May 2008. ACM Press.

[16] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[17] Jens R. Calamé, Natalia Ioustinova, and Jaco van de Pol. Automatic Model-Based Generation of Parameterized Test Cases Using Data Abstraction. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proceedings of the Doctoral Symposium affiliated with the Fifth Integrated Formal Methods Conference (IFM 2005)*, volume 191 of *Electronic Notes in Computer Science*, pages 25–48. Elsevier, October 2007.

[18] Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli, and Mounir Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *DS-RT '08: Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 315–318, Washington, DC, USA, 2008. IEEE Computer Society.

[19] Yanping Chen, Robert L. Probert, and Hasan Ural. Model-based regression test suite generation using dependence analysis. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 54–62, New York, NY, USA, 2007. ACM.

[20] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: a system for selective regression testing. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 211–220, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[21] S. Colin, B. Legeard, and F. Peureux. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):213–235, 2004.

[22] Frédéric Dadeau and Régis Tissot. jsynopsys – a scenario-based testing tool based on the symbolic animation of b machines. *Electron. Notes Theor. Comput. Sci.*, 253(2):117–132, 2009.

[23] David Delmas and Jean Souyris. Astrée: From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[24] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[25] Wenliang Du and Aditya P. Mathur. Testing for software vulnerability using environment perturbation. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 603–612, Washington, DC, USA, 2000. IEEE Computer Society.

[26] Bruno Dutertre and Maria Sorea. Timed systems in sal. Technical report, Computer Science Laboratory, 2004.

[27] The eclipse foundation, eclipse test and performance tool platform project.

[28] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, 2000.

[29] Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology.* 1990.

[30] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for Model-based Requirements Testing of Service Oriented Systems. In *Proceedings of the IASTED International Conference*, volume 642, page 018, 2009.

[31] M. Felderer, F. Fiedler, P. Zech, , and R. Breu. Flexible Test Code Generation for Service Oriented Systems. 2009. QSIC'2009.

[32] Michael Felderer, Berthold Agreiter, Ruth Breu, and Alvaro Armenteros. Security testing by telling teststories. In Gregor Engels, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2010*, volume 161 of *LNI*, pages 195–202. GI, 2010.

[33] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp - a protocol validation and verification toolbox. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, London, UK, 1996. Springer-Verlag.

[34] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[35] Martin Fränzle and Christian Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Form. Methods Syst. Des.*, 30(3):179–198, 2007.

[36] Gordon Fraser, Bernhard K. Aichernig, and Franz Wotawa. Handling model changes: Regression testing and test-suite update with model-checkers. *Electr. Notes Theor. Comput. Sci.*, 190(2):33–46, 2007.

[37] Gordon Fraser and Franz Wotawa. Property relevant software testing with model-checkers. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006.

[38] Gordon Fraser and Franz Wotawa. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *ICSEA '06: Proceedings of the International Conference on Software Engineering Advances*, page 16, Washington, DC, USA, 2006. IEEE Computer Society.

[39] Gallio automation platform.

[40] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, 1999.

[41] M.-C. Gaudel and P. R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 10(5-6):436–451, 1998.

[42] Arnaud Gotlieb. Euclide: A constraint-based testing framework for critical c programs. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:151–160, 2009.

[43] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems*, pages 209–212, London, UK, 2002. Springer-Verlag.

[44] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[45] Paul Le Guernic, Paul Le Guernic, Jean-Pierre Talpin, Jean pierre Talpin, Jean-Christophe Le Lann, Jean christophe Le Lann, and Projet Espresso. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.

[46] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[47] David Harel and P. S. Thiagarajan. Message sequence charts. pages 77–105, 2003.

[48] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[49] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[50] Ibm rational quality manager.

[51] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, 2004.

[52] Claude Jard and Thierry J&#x00e9;ron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[53] Dennis Jeffrey and Neelam Gupta. Test suite reduction with selective redundancy. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 549–558, Washington, DC, USA, 2005. IEEE Computer Society.

[54] Thierry Jéron. Symbolic model-based test selection. *Electron. Notes Theor. Comput. Sci.*, 240:167–184, 2009.

[55] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[56] Jacques Julliand, Pierre-Alain Masson, and Régis Tissot. Generating security tests in addition to functional tests. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 41–44, Leipzig, Germany, May 2008. ACM Press.

[57] Jacques Julliand, Pierre-Alain Masson, and Régis Tissot. Generating tests from B specifications and test purposes. In *ABZ'2008, Int. Conf. on ASM, B and Z*, volume 5238 of *LNCS*, pages 139–152, London, UK, September 2008. Springer.

[58] Jan Jürjens. Umlsec: Extending uml for secure systems development. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 412–425, London, UK, 2002. Springer-Verlag.

[59] Jan Jürjens. Model-based security testing using umlsec. *Electron. Notes Theor. Comput. Sci.*, 220(1):93–104, 2008.

[60] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *Computer Software*, pages 200–204. Springer, 2002.

[61] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell, 1997.

[62] Yves Ledru, Lydie du Bousquet, Olivier Maury, and Pierre Bontron. Filtering tobias combinatorial test suites. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2004.

[63] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43:306–320, 1994.

[64] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *Proc. of the Int. Conf. on Formal Methods Europe, FME'02*, volume 2391 of *LNCS*, pages 21–40, Copenhaguen, Denmark, July 2002. Springer.

[65] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions: Gatel. In *ASE*, pages 229–, 2000.

[66] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[67] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, pages 119–131, Las Vegas, USA, October 1999. IEEE Computer Society Press.

[68] I. Parissis. A Formal Approach to Testing LUSTRE Specifications. In *Proceedings of the 1st International IEEE Conference on Formal Engineering Methods*, Hiroshima, Japan, 1997.

[69] S. J. Prowell. Jumbl: A tool for model-based statistical testing. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 337.3, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Wolfgang Reisig. *Petri nets: an introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[71] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, 1996.

[72] The RT builder web site.

[73] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education, 2004.

[74] V. Rusu. Combining formal verification and conformance testing for validating reactive systems. *Journal of Software Testing, Verification, and Reliability*, 13(3), September 2003.

[75] Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 171–187, Berlin, Heidelberg, 2009. Springer-Verlag.

[76] J. M. Spivey. *The Z notation: a reference manual.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[77] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27(4):97–106, 2002.

[78] Swedish Institute of Computer Sciences. *SICStus Prolog 3.11.2 manual documents*, June 2004. http://www.sics.se/sicstus.html.

[79] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing security policies: Going beyond functional testing. In *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pages 93–102, Washington, DC, USA, 2007. IEEE Computer Society.

[80] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[81] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science, 2006.

[82] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[83] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical report, April 2006.

[84] B. Vaysburg. Model based regression test reduction using dependence analysis. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 214, Washington, DC, USA, 2002. IEEE Computer Society.

[85] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[86] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, page 264, Washington, DC, USA, 1997. IEEE Computer Society.

[87] Lihua Xu, Marcio Dias, and Debra Richardson. Generating regression tests via model checking. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference*, pages 336–341, Washington, DC, USA, 2004. IEEE Computer Society.

[88] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, New York, NY, USA, 2006. ACM.

[89] Justyna Zander, Zhen Ru Dai, Ina Schieferdecker, and George Din. From u2tp models to executable tests with ttcn-3 - an approach to model driven testing. In Ferhat Khendek and Rachida Dssouli, editors, *Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005*, volume 3502 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2005.