D7.1: Evaluation of existing methods and principles

Fabrice Bouquet (INR), Frederic Dadeau (INR), Pierre-Alain Masson (INR), Zoltán Micksei (BME), Berthold Agreiter (UIB), Bruno Legeard (SMA), Michael Felderer (UIB), Stéphane Debricon (INR), Daniel Varro' (BME), 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	6.0
Status	Final
Work Package	WP 7
Deliverable Type	Report
Contractual Date of Delivery	M6
Actual Date of Delivery	01 September 2009
Responsible Unit	INR
Contributors	F. Bouquet, F. Dadeau, P.-A. Masson, Z. Micksei, B. Agreiter, B.Legeard, M. Felderer, S. Debricon, D.Varro', E.Chiarani, J.Jurjens, F. Paci, F. Massacci,
Keyword List	Model Based Testing
Dissemination level	PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	26 June 2009	Draft	F. Bouquet (INR)	First version
0.2	2 July 2009	Working	F. Dadeau (INR)	Paragraph 4.1
0.3	2 July 2009	Working	PA. Masson (INR)	Paragraph 3.2
0.4	22 July 2009	Working	Z. Micksei (BME)	Section 5
1.0	22 July 2009	Draft	B. Agreiter (UIB)	Review
1.1	27 July 2009	Working	D. Varro' (BME), E. Chiarani, F. Paci (UNITN)	Quality check
2.0	29 July 2009	Working	B. Legeard (SMA)	Review
2.1	29 July 2009	Working	J. Jurjens (OU/TUD), E. Chiarani (UNITN)	Quality check
3.0	30 July 2009	Draft	F. Bouquet (INR)	Review
3.1	30 July 2009	Working	J. Jurjens (OU/TUD), F. Massacci, E. Chiarani (UNITN)	Quality check
3.2	13 August 2009	Working	B. Agreiter, M. Felderer (UIB)	Section 2.1 and review
3.3	23 August 2009	Working	F. Bouquet (INR)	Review
4.0	25 August 2009	Draft	F. Bouquet, PA Masson (INR)	Review
4.1	25 August 2009	Working	E. Chiarani (UNITN)	Quality check
5.0	26 August 2009	Draft	F. Bouquet (INR)	Minor edits + revised version
5.1	25 August 2009	Working	E. Chiarani (UNITN)	Minor edits + revised version
5.2	31 August 2009	Working	F. Bouquet (INR)	Minor edits

6.0	31 August	Final	J.Jurjen (OU/TUD), E. Chiarani (UNITN)	Finalised version

Executive summary

This document provides a state-of-the-art overview on the Model-Based Testing (MBT) methods. The aim of this work package is to develop a method that can be used for security testing and takes evolution of a system into account. This document explains several elements that can be used to develop such a methodology. Tests are generated by a model-based approach. Test sets are based on particular coverage criteria, e.g. security requirements criteria.

Following the Secure Change vision, we will propose a methodology based on the evolution of the requirements. We propose to rely on two test sets; the first one is computed from the model of security requirements before evolution, and the second one is computed from security requirements after the evolution. Parts of these tests will be dedicated to non-regression testing, whereas new tests will be generated in order to exercise the evolved requirements. We will develop a tool-supported methodology that is able to compute test cases and to determine the conformance of an evolvable system w.r.t. security requirements expressed through formal models.

The document at hand consists of four parts. After an introduction on MBT, the modeling language for MBT is presented. In this chapter we further discuss associated tools.

The second part presents selection and coverage criteria used in MBT. Such criteria are a very important element of a test approach because the test method explores a sub-part of the implementation or system under test and the criteria can give the degree of confidence in the generated tests.

The third part discusses model-based security tests. This chapter introduces work about how security requirements can be tested in Secure Change.

The last chapter presents regression with MBT. The regression test can be seen like the first step to establish an approach for validation of the evolution as proposed in the Secure Change project.

Index

1 Introduction Approach MBT

This chapter introduces concepts associated with the Model-Based Testing (MBT). Each element used or necessary for MBT will be identified and discussed.

Modern software development has started to make use of models extensively, be it for a formal representation of the specification, or for the generation of code. As proposed in [51] we can distinguish between at least four different approaches for making use of models in the testing process. The authors begin their matter on the importance of modeling for the validation of a system. The question is where models originate from and what are they used for. The authors propose four alternatives:

1. The same model is used for test case generation and for code generation of the actual system under test (SUT).
2. The SUT is implemented manually out of the specification and the model is extracted in an automatic way from the implementation. It is then used in order to generate the tests
3. The model is created manually starting from the system specification and it is then used to generate tests.
4. Two distinct models are used, one to generate tests and one to generate code for the implementation.

In fact, the first scenario has two problems. One is due to the lack of redundancy between the expected behavior of a system (model) and its actual behavior (implementation). This redundant information is crucial to establish a verdict. The other problem is due to constraints on the level of abstraction, the model should be very close to the implementation level to allow the generation of code.

The second scenario also has the problem of lack of redundant information. The third and fourth scenarios provide the best conditions for verifying the quality of the SUT.

For Secure Change the third and fourth scenarios, which consider a series of validations in which separate models are used for test generation, seem to be the most promising ones. The idea is to formalize the specification and create an abstracted view of the system. This abstract view allows the usage of dedicated tools to automate the production and the execution of tests. These scenarios are also the optimal ones according to [55] and [51]. In both cases the views of software developers and test engineers are presented in a separated way.

In fact, the fourth scenario can be mixed with the first one. It is possible to have one single model, which is split into two separate views to establish a strong link between the implementation API and the model used for test generation.

One of the major research areas in model-based testing is the automatic generation of test sequences. Test sequences must be executed on the implementation under test (IUT). Tools such as [68] or [94] (now Leirios marketed as Smartesting), [76] or [82],

work on models [50] in the form of concurrent automata like Statemate or UML statecharts with OCL or in the form of Pre/Post models like B or Z and generate tests which guarantee satisfaction of well-defined model coverage criteria such as the coverage of all transitions or states in the model, the coverage of effects, or the coverage of inputs. These techniques are based on coverage criteria of models and do not necessarily correspond to coverage criteria at the code level.

A second approach also takes into account more dynamic criteria. The model is completed by other information, such as properties or schema that are used in the definition of so-called property-based coverage criteria.

For example, a coverage criterion can be based on covering the test purpose or target that has to be defined by the user, usually in a similar formalism to the model of the implementation under test (IUT). Sometimes the user has to model the environment of the IUT. The tool combines the model of the IUT with the test purpose and/or environment to automatically generate tests focusing on behaviour specified by the purpose. The test purpose or environment can be seen as another form of test model. Such tools have been developed for several modeling paradigm as in [86] and [96] for transition systems, [87] for transition systems extended with data, [69] (currently being integrated into the Scade tool) and the [100] tool for Lustre.

Model-based statistical testing (MBST) was also proposed as a black-box test technique that enables the generation of representative tests from the tester's or user's perspective [44]. In MBST a so-called usage model is built to describe the relevant system inputs, input sequences, and system responses as a discrete time Markov chain, a state machine annotated with transition probabilities.

A specific feature of MBST is the estimation of test object reliability based on the test cases executed and test results [9]. Further extensions have been developed to define equivalence classes for the test oracles and incorporate requirements coverage of usage models and test cases. MBST is supported by academic [93] and commercial [90] tools.

2 Modeling for test

In this chapter, we present an overview of several techniques using models in the testing process. We focus on the scenarios three and four introduced in the previous chapter because they have been considered as the most promising ones. We discuss one approach for scenario four, where two distinct models are used, one for testing and one for system modeling (Telling TestStories). The other approaches in this chapter represent scenario three, where the test model is extracted from the system specification and used for generating test cases (Transition/State Models, Symbolic Transition/State Models and Pre/Post Models).

2.1 Telling TestStories

Model-driven testing is based on the derivation of executable test code from models in a MDA manner [64]. 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) [24] 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 1.

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.

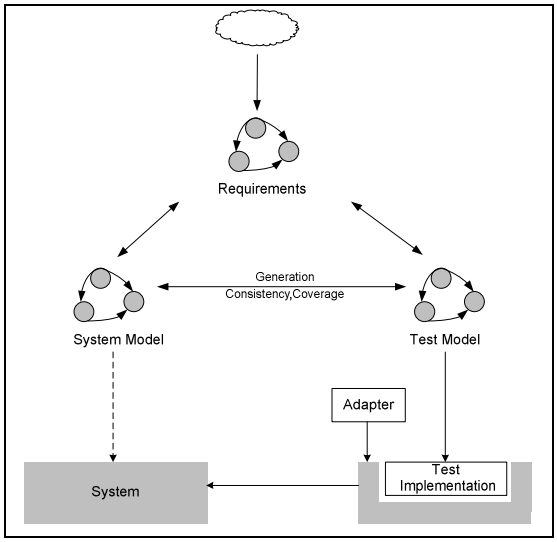

Figure 1: Overview of TTS Artefacts

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 [26]. 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.

2.2 Transition/State Models

In this section, we address the conformance testing for systems modelled by using labelled transition systems (LTS or -- Labelled Transition System). We are mainly interested in the transition systems labelled with inputs and outputs [57] (also known IOLTS -- Input / Output Labelled Transition System) that are more widely used in conformity testing. Compared with the LTS, the IOLTS are particularly well suited to model reactive system, because they distinguish controllable actions from observable actions.

In the following we clarify some notions and notations on systems transitions labelled by enrichment of the distinction input / output and the concept of quiescence. Then, we explore relationship with Ioco (Input-output conformance) used to determine the verdict conformity between the model and the system under test. Then we define the notion of test cases.

The formalism of labelled transition systems (def. 1) allows to model a system in the form of sets of statements (including in particular the initial state of the system), name of actions (among which we can distinguish the internal action τ) and transitions. The modeling of a system and its environment can be described by a LTS, but in this case, there is no difference between emissions and receptions. Since for testing purposes, it is important to separate between what can be controlled and what can be observed, the enrichment of LTS in IOLTS (def. 2) by a signature action is useful. It may be noted that sets of action names input and output are separated.

Definition 1 (Labelled Transition system – LTS). *A LTS is defined by a quadruplet* $< Q, q_0, A, \rightarrow >$ *where:*

- Q *is a bounded set of non empty state,*
- $q_0 \in Q$ *is the initial state of system,*
- $A \cup \{\tau\}$ *is a set of action terms with specific action* $\tau \notin A$. τ *characterized all internal action (unobservable). We noted* $A_\tau = A \cup \{\tau\}$
- $\rightarrow \subseteq Q \times A_\tau \times Q$ *transition relation.*

Definition 2 (Input Output Labelled Transition system – IOLTS). *An IOLTS is defined by a quadruplet* $< Q, q_0, A, \rightarrow >$ *where:*

- Q *is a bounded set of non empty state,*
- $q_0 \in Q$ *is the initial state of system,*

- *$A \cup \{\square\}$ is a set of action terms with specific action $\tau \notin A$. τ characterized all internal action (unobservable). $A = A_I \cup A_O$ and $A_I \cap A_O = \varnothing$ with A_I set of all input actions (received) and A_O set of all output actions (sending). Input and output actions can be activate from any state.*
- *$\rightarrow \subseteq Q \times A_\tau \times Q$ transition relation.*

The notion of quiescence (def. 3) explains normal blocking situations of the system. Among these blocking situations, we can distinguish:

- The blocking of exit (or output lock): this is a deadlock in a state from which no exit action and no internal action are specified.
- The "deadlock": these blocking situations arise in a state from which no transition is activated.
- The "livelock": these situations arise when there is a path (not empty) of internal actions from a state that leads to the same state.

This quiescence is represented by the special action δ. In practice, observation of quiescence in the execution of a test case is performed at the expiration of a timer.

***Definition 3 (Quiescence)**. As an IOLTS < Q, q_0, A, $\rightarrow$ >, a state $q \in Q$ is called quiescence if there is no transition of A_τ activable from the state q.*

To compare the system under test to its model, the formal relationship can be verified with all the correct implementation with regard to the model. The ioco conformance relation [57] is based on inclusion of traces. An implementation I conforms to its specification S, if for every trace σ of S, the system under test produces outputs and quiescence included in the model. A test case and IOLTS statements are marked by Pass, Fail or Inconclusive. This marking will set the verdict. Running a test case is conducted by parallel composition of the latter with the system under test. The verdict is given by the marking of the status achieved by a complete trace of the parallel composition of the test case and system under test.

To be complete, the generation of test cases should consist of extracting all traces of the model leading to a deadlock situation. In practice, we are quickly confronted with situations of combinatorial explosion due to the number of statements, branches and loops.

The finite state machines are very close to the labelled transition systems. As shown in definition 4, the difference lies in the fact that the sets of states and transitions are finite and that all statements of acceptance are used to determine the correct execution of the system.

***Definition 4 (Finite Sate Machine - FSM).** A Finite State Machine is a quintuplet < S, Σ, δ, s_0, F >, where:*

- *S is a finite set of states,*
- *Σ is a finite set of actions,*

- $\delta; S \times \Sigma \rightarrow S$ *a transition function,*
- $s_0 \in S$ *is the initial state,*
- $F \in S$ *is a set of all final states.*

The Unified Modeling Language (UML) is widely used as a modeling support for model-based testing. There are several reasons for this interest. First, UML provides a large set of diagrammatic notations for modeling purposes, with several complementary representations. A static representation (i.e. class diagrams) is used to model the points of control and observation of the IUT and the data that represent the abstract state of the IUT. A dynamic representation (e.g. state diagrams or activity diagrams) is used to model the expected behaviour of the IUT. Second, the Object Constraint Language (OCL [60]) associated with UML makes it possible to have precise models – this means that the expected behaviour can be formalized using OCL. Third, UML is the de-facto industrial standard for modeling enterprise IT applications; most software engineers have had some first level training on UML – this is an important point to facilitate the acceptance of a disruptive process such as model-based testing.

However, UML contains a large set of diagrams and notations, defined in a flexible and open-ended way using a meta-model and allows for different interpretations of the semantics of the diagrams by different UML tools. So for practical model-based testing it is necessary to select a subset of UML and clarify the semantics of the chosen subset so that model-based testing tools can interpret the UML models.

There are numerous model-based testing approaches that use UML as a modeling notation. Some of them are based on sequence or interaction diagrams to express scenarios (see e.g. [11]), state machines to express behaviour models (see e.g. [59]) or combine them (see e.g. [17]). Few approaches are using OCL as an action language for model-based testing. B. K. Aichernig proposes an approach based on mutation analysis of OCL specifications [13], and [10] proposes a combination of test cases using very similar approach. But currently, there is no subset of UML/OCL clearly proposed for model-based testing, except in [15]. This subset is UML/OCL used by [94] model-based testing tool.

2.3 Symbolic Transition/State Models

Symbolic Labelled Transitions Systems (SLTS) or STS (Symbolic Transition System) is an extension of labelled transition systems taking into account the explicit data flow. The definition of an STS is given in definition 5, [31], shows that the enrichment-related assets in STS are the introduction of:

- State variables,

- Parameters of action,

- Guards on transitions (expressed on state variables and parameters)

- Terms expressing changes of state variables during transitions.

***Definition 5 (Symbolic Transition System - STS)**. A Symbolic Transition System is 7-uplet < L, l_0, V, t, I, A, →>, where:*

- *L is a finite set of states,*
- *l_0 is the initial state,*
- *V is the set of state variables*
- *I is the set of parameters,*
- *A is the set of action,*
- *$\rightarrow \subseteq L \times A \times P(V \cup I) \times T(V \cup I)^V$ is transition relation, with $P(V \cup I)$ a set of first order logic formula on the state variables and parameters (guard of action) and $T(V \cup I)^V$ a set of terms on the state variables and parameters describe modification of state variables by the action.*

The semantics of STS is given in terms of LTS [31, 45]. The use of labelled transition systems for generating symbolic tests requires the use of other techniques than a simple path finding in a reachability graph. For example, the symbolic animation of the model can extract symbolic test cases that will be instantiated by a solver.

2.4 Pre/Post Models

The B language [1] is a modeling language based on the logic and the set theory. A system is modeled as an abstract machine (or a set of abstract machines as part of the composition of machines and the process of refinement). A B machine consists of various clauses including:

SETS: declaration sets listed

CONSTANTS: declaration of constants

PROPERTIES: properties on constants

VARIABLES: declaration of variables

INVARIANT: definition of invariant properties on the system

ASSERTIONS definition properties arising from the invariant

INITIALIZATION: initialization of variables

OPERATIONS: Operations

The system actions are defined by operations in the form of substitutions. Among the operators of substitutions, we find substitutions kept (ANY, SELECT), substitutions of choice deterministic (IF, CASE) or indeterminate (CHOICE), substitutions of assignment (:=, :□), without the substitution effect (skip) and multiple substitutions (||), which allows substitution of composing two in parallel. The B language was introduced as part of the method B [1] that aims at the development of software, from its specifications, in successive stages of refinement and proof. Firstly, the system is formalized at a high abstraction level, and then it is refined by successive stages to reach the abstraction level of the implementation from which the code of the implementation can be produced. During this process, the proof allows to:

- Check the consistency of a machine, by proving that the invariant is preserved by the substitutions of the initialization and of the operations,,

- Check the refinement, which means proving the a machine is a correct refinement of another (more abstract) one.

As a consequence, many tools have been developed around the B language, including model-checkers, that allow for test generation from B models. Some of these tools are presented in the following sections.

The JML [42] is an annotation language for Java programs. It allows to model specification in a program in the form of annotations that can be:

- Pre-conditions and post-conditions on the functions and methods,

- Class invariants,

- Loop invariants.

The JML is based on a concept of design by contract, whose main idea is that interactions between a class and its clients are governed by contracts expressed by properties. Thus, the "client" guarantees the respect of certain constraints when calling a method of a class (respect of the preconditions of the method). This guarantees the respect of some class properties (expressed as post-conditions) in return. The properties are expressed in first-order logic on the attributes of the class and the parameters of methods or functions. For example, Figure 1 shows the JML annotations to a method of swapping two values in an array of integer (tab [i] and tab [i + 1]):

- Requires the clause gives the pre-conditions of the method,

- Ensure the clause gives the post-conditions of the method,

- Assignable clause gives variables whose value can be changed.

```
/*@ requires tab != null && tab.length > 1 && i>=0 && i+1<tab.length;
@ ensures (\forall int j; ((j>=0 && j<i)||(j>i+1 && j<tab.length)) ==> tab[j]==\old(tab[j]))
@ && tab[i]==\old(tab[i+1]) && tab[i+1]==\old(tab[i]);
@ assignable tab[i],tab[i+1];
@*/
void permut(int[] tab, int i){
int temp0 = tab[i];
tab[i] = tab[i+1];
tab[i+1] = temp0;
}
```

Figure 2. Example of JML annotation

Many tools have been developed for JML, the best-known being:

- Jmlc: a compiler that can compile JML annotations in the Java bytecode enabling assertion checking at runtime,

- The unit testing tool that combines jmlunit compiler JML and JUnit tool to use JML annotations for the preparation of verdicts,

- The static verification tool escjava2, which uses JML annotations to detect errors,

- The Key proverb based sequent calculus that is used to generate proof obligations from JML annotations and prove them in an automatic or semi-automatic manner.

Furthermore, we cite two tools for test generation from formal specifications in JML: JML-TT and Jartege.

ASML languages [7, 8] and spec # [19] are two modeling languages based on the ASM (Abstract State Machine). The ASM is based on the description of statements and symbolic functions of transformation of these states. Spec # model takes the form of annotations in a C #. It allows defining a DSO through the formalization of pre-conditions and post-conditions. The tool for test generation specExplorer supports these languages

3 Test Selection and Coverage Criteria

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 IUT. We identified two kinds of criteria to insure coverage.

3.1 Static Criteria

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

3.1.1 Control flow graph criteria

To adopt a method for structural testing based on the cover of the graph control is to propose a certain set of paths on a graph in order to form tests campaign. Satisfy a structural testing method for a given coverage is therefore to find tests that enhance control paths (i.e. paths of execution) and covering paths provided by the method adopted. Under the criteria based on the graph control, there are many criteria for coverage:

- 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 lower criterion to test to the highest (exception made of the cover paths and internal limits that 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 follow [58]:

- **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 others tests. Decisions are the branch criteria that modify the flow of control in selection and interaction statement.
- **Path Coverage (PC)**: test suite must execute ever 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 diction 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 [48], for code-based coverage we have PC → DC → SC, where C_1 → C_2 indicates that every test suites satisfies C_1 also satisfies C_2.

More generally as propose in [58], fig. 3 give hierarchy between criteria.

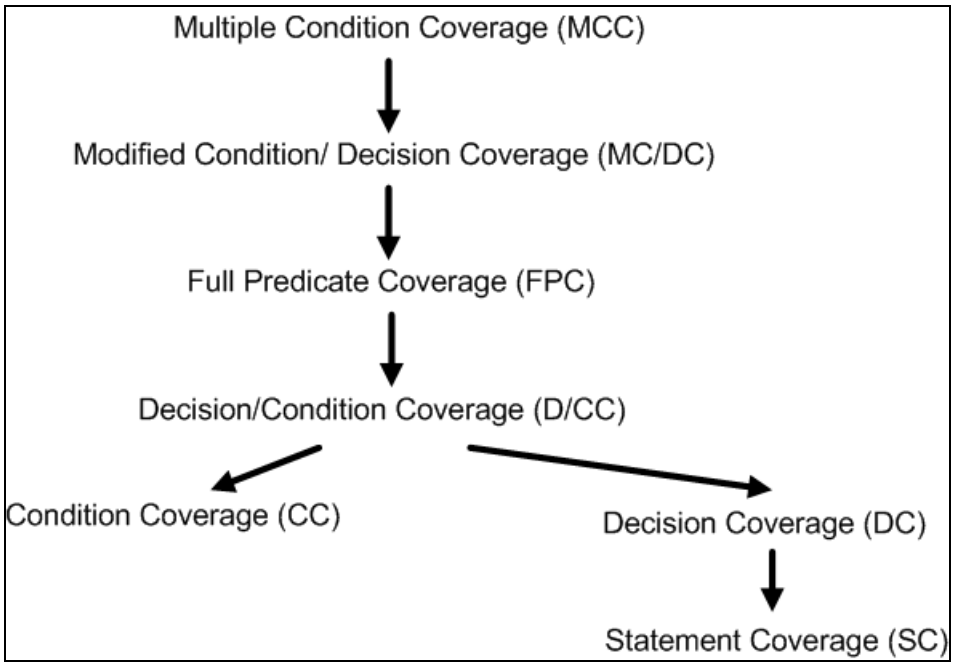

Figure 3 – The hierarchy of control flow coverage criteria

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.

3.1.2 Data flow criteria

Control 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:

All-def-use-paths → All-uses → All-defs

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

3.2 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 [4], 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 [18] and [35], or as a regular expression in [43].

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 labelled by action names.

In our approach, the validation engineer manually describes by means of a test purpose TP (see Def. [TP]) how he intends to test the system, according to his know-how. We have proposed in [37] 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 [TP] (Test Purpose). *A test purpose on a model M (with a set O^M of operations) is a tuple $<Q^P , q_0^P, T^P , \lambda^P , Q^P, Q_f^P>$ 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 and $\lambda^P \subseteq Q^P \rightarrow Pred^M$ is a total function that associates with each state q a state predicate denoted as $\lambda^P(q)$ ($\subseteq Pred^M$).*

In [38], 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 IUT. This approach has been successfully experimented on the industrial application IAS (Identification Authentication and Signature) with Gemalto.

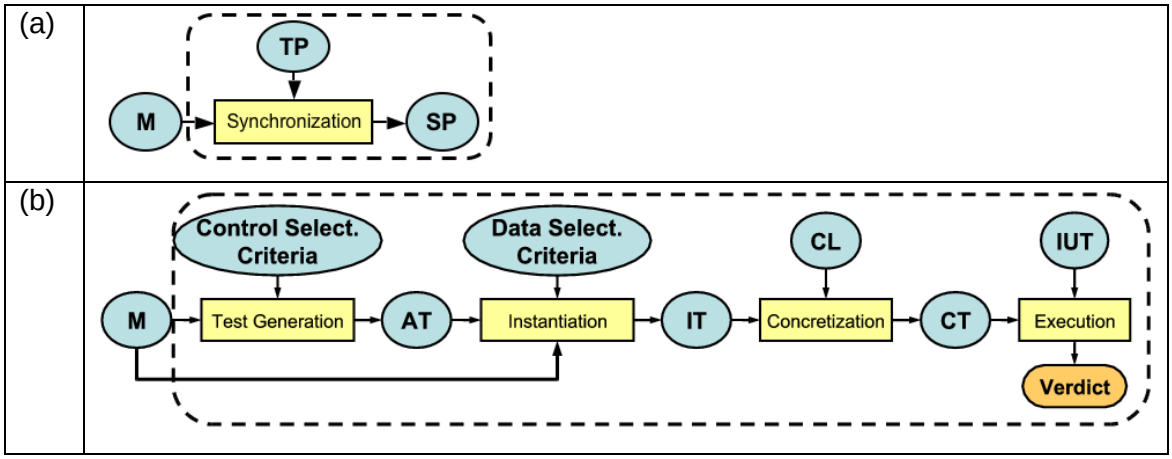

Figure 4 – Generating tests from dynamic selection criteria

Our approach is depicted in two views in Fig. 4. View 4(b) shows the usual MBT process with static (structural) selection criteria. The MBT process with dynamic criteria is obtained by replacing the input M of View 4(b) by the output SP of View 4(a).

View 4(b). From a behavioural model M written by the validation engineer and from static selection criteria on the control structures of the model, the test generation tool computes a set of abstract tests AT. The instantiation computes parameter values for the operation calls, providing a set of instantiated tests IT. This relies on data selection criteria such as boundary valuation strategies. These tests are still as abstract as M and must be concretized via a concretization layer CL into a set of concrete tests CT, that are executed on the IUT. This delivers a verdict of success or failure, by comparing the results predicted by M with those returned by the IUT.

View 4(a). The input M in View 4(b) is replaced with a model SP that results from the synchronization of M with a dynamic selection criteria TP. Thus executions of SP are executions of M that match TP.

3.3 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 [84], Uppaal [88], CADP [77] 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 [65] solvers to perform bounded model checking on infinite-state systems (SAL) [73], hybrid systems (HySAT [74], HyTech [83]) or Lustre (Prover).

Another direction is probabilistic model-checking with tools like PRISM [92] and statistical model-checking [22] which 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 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 [16]. They are usually real-time so models with an explicit representation of time, or at least the order in which discrete events take place, are necessary to represent timing properties. 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 [54]. 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 [102] and methodologies to test embedded software with hardware in the loop. The model of computation introduced with the Signal synchronous language [67], then further developed in the Polychrony [89] workbench and industrialized in RT-Builder [101], 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.

The first question addressed by this project is the relationship between the specialized models used in model-based V&V and more global models used for model-based design and development, including the question of how to obtain V&V models from requirements, specifications and design and development models. Model-based V&V technologies also pose the second problem addressed by this project: the link between the model and the implementation that is necessary to ensure that properties verified/validated on the model are satisfied by the implementation. Although automatic code generation was supposed to eliminate this problem, we see that on the one hand automatic code generation does not always guarantee that the semantics of the model and the implementation are identical in all respects and on the other hand, the implementation often includes portions of code which have had to be developed by hand or which correspond to "library" or "external" components of the model whose

semantics is left implicit. This second problem will be addressed with the aid of a second class of V&V tools which have attracted a lot of research effort and which are now relatively mature. These are the tools that work on the source code of the implementation, either analyzing it statically or generating tests to cover objectives (such as coverage criteria) defined on the implementation.

4 MBT for Security

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.

4.1 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 (IUT). A formal verification of the IUT is usually out of reach due to its impracticable size.

Our idea is to use the security properties as test purposes, i.e. as dynamic selection criteria, to guide the test generation. We have exposed in [46] that this is a mean to automatically generate security tests from the model.

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 [30], 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's structure, with its oracle. It can automatically be converted into a verifiable model [3]. 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.

4.2 Decision of Validity or Invalidity

Deciding about the validity of tests in the old test suite can be obtained by symbolically animating all the tests on the new model. Tests for which the results have changed or that can no longer be animated are the invalid ones. The others are valid. With a model-checker, this result can be obtained by model checking a special temporal property on the test case model. The property claims that output values of the test case and the new model are equal for the length of the test case. Each test case that results in a counter example is invalid. Remaining tests are still valid, which means that they are not affected by the model change. The drawback of this approach is that the whole new model is involved in model-checking.

To avoid this, the validity or invalidity of a test case can preferably be obtained by checking its model against some temporal properties called *change properties*. They can be created automatically from the model source file. If a transition condition or target is changed, a test case is still valid if it fulfils the property

$G(changed\_condition \Rightarrow X\ variable=changed\_value)$.

This property claims that on a changed condition, the value of a variable after the transition equals the changed value.

If a variable transition is removed, it can only be determined whether a test case takes the old transition using a negated property

$G(old\_condition \Rightarrow X\ \neg(variable = old\_value))$.

This property claims that on an old condition, the value of a variable after the transition does not equal the changed value. Any test case that takes the old condition results in a counter example.

4.2.1 Creation of the New Tests

For the creation of the new tests, 3 methods are proposed in [30].

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* [32]. It is a generalisation of [62] 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 – 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.

As we have seen, this approach is based on state-transition models and temporal properties, and it makes use of model-checkers. Querying a model-checker with a temporal property has some similarities with animating a test purpose on a behavioural model, in the sense that it computes selected traces by means of a dynamic selection criterion. So we wonder if the approach of [30] could be adapted to our framework, where we use pre-post models instead of state-transitions ones, and where we perform symbolic animation rather than model-checking.

4.3 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 5 illustrates this principle.

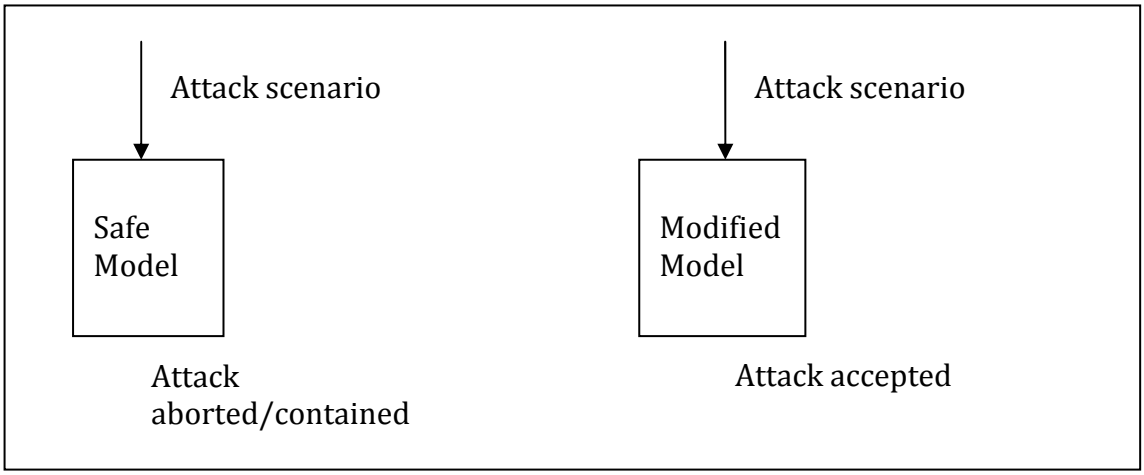

Figure 5. Model modification to accept attacks

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 IUT, if this latter reacts as the modified model, then the IUT presents a weakness.

This approach is called mutation-based testing [49]. 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 [28]. [29] present notions of relevance of test cases w.r.t. (possibly security) properties, based on their error detection capabilities.

In [47], 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 IUT.

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 [23]. This kind of mutation can be classified as invasive because it involves non-functional aspects of the IUT.

A related work has be done by [40], based on fault-injection techniques. The idea is to introduce security errors in UMLsec models [39], 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 [5].

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 [70] tool.

5 Regression Testing

Changes in software's artefacts throughout its lifecycle could make previously fixed bugs re-appear or brake existing functionality [14]; 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 [12].

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" [33]. 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 [34]. 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.

5.1 Regression testing techniques

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

a) Regression test selection: select only tests from the regression test suite that are affected by changes.

b) 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.

c) Coverage identification: identify those parts of the system that need additional tests due to the change.

d) 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.

e) 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 [52]. 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 [36]. 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 [63] to uncover changed portions not touched by existing tests. More advanced approaches typically use some sophisticated data structure, e.g. program dependence graphs [27] 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 [53] 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.

5.2 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 [85, 91] and open source ones [95] [75]) 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 [71] 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) [61, 98] 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 [56] 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.

5.3 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 [72, 79, 80, 81, 83], 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 [20] 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 of [41] works similarly on EFSM models, and its focus is to reduce an existing regression test suite based on dependency analysis.

6 Conclusion

We present several approaches of MBT. Modeling language, coverage criteria and objective of the tests (security and evolution) compose the MBT approach.

We decide to use UML language as modeling language. So, we can develop a tool-supported methodology that is able to compute test cases and to decide the conformance of an evolvable system w.r.t. security requirements expressed through evolvable formal models.

So, our framework can be able to identify evolutions in the formal model, isolate non-impacted requirements and their corresponding tests, and compute new test cases for the evolved will concretize these parts of the software. For the security part, we propose to use a dedicated test purpose to explain security properties.

For work-package 7, the next step of the work is to prepare a dedicated approach based on this existing works presented in this document. So, we will study the impact of the evolution on model based testing approach in account of case studies

7 References

[1] Abrial, J.R *The B Book, Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996 (ISBN 0521496195)

[2] Aichernig, B.A, Pari Salas P.A, Test Case Generation by OCL Mutation and Constraint Solving. QSIC 2005: 64-71

[3] Ammann, P. and Black P.E., A specification-based coverage metric to evaluate test sets. In HASE'99, 4th int. symposium on High-Assurance System Engineering, pp. 239-248, 1999.

[4] Ammann, P., Black, P.E, and Majurski, W., Using model checking to generate tests from specifications. In ICFEM'98, pages 46–54. IEEE, 1998.

[5] Aslam, T., Krsul, I. and Spafford, E.. Use of a taxonomy of security faults. In 19th National Information Systems Security Conference (NISSC), pages 551-560, 1996.

[6] Ball, T., Podelski, A. and Rajamani S., Boolean and Cartesian Abstraction for Model Checking C Programs. In Proceedings of TACAS: Tools and Algorithms for the Construction and Analysis of Systems. Genova, Italy, April 2001.

[7] Barnett, M., Grieskamp W., Nachmanson L., Schulte W., Tillmann N., and Veanes M., Model-based testing with asml.net. In 1st European Conference on Model-Driven Software Engineering, December 2003.

[8] Barnett, M., Grieskamp W., Nachmanson L., Schulte W., Tillmann N., and Veanes M., Towards a tool environment for model-based testing with AsmL, volume 2931/2004. 2004.

[9] Bauer, T., Böhr F., Landmann, D., Beletski, T., Eschbach, R. and Poore, J.H., From Requirements to Statistical Testing of Embedded Systems. Software Engineering for Automotive Systems - SEAS 2007, ICSE Workshops, Minneapolis, USA.

[10] Benattou, M., Bruel, J.-M., and Hameurlain, N., "Generating Test Data from OCL Specification" in Proceedings of the ECOOP'2002 Workshop on Integration and Transformation of UML models (WITUML'02), 2002.

[11] Beyer, M., Dulz, W., Zhen, F., "Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains," ats, p. 102, 12th Asian Test Symposium (ATS'03), 2003

[12] Beizer, B., Software Testing Techniques. John Wiley & Sons, Inc. 1990.

[13] [Bernahard et al. 05] Bernhard K. Aichernig, Percy Antonio Pari Salas: Test Case Generation by OCL Mutation and Constraint Solving. QSIC 2005: 64-71

[14] Binder, R. V., Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc. 1999.

[15] Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N. and Utting M., A subset of precise UML for model-based testing. In A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing, London, UK, pages 95--104, July 2007. ACM Press. Note: A-MOST'O7 is colocated with ISSTA 2007, Int. Symposium on Software Testing and Analysis.

[16] Brandl, H., Aichernig, B. K., and Wotawa, F., Conformance Testing of Hybrid Systems with Qualitative Reasoning Models. MBT 2009, Fifth Workshop on Model-Based Testing, March 22, 2009, York,UK

[17] Briand, L., Labiche, Y.,A UML-Based Approach to System Testing, Proceedings of the Fourth International Conference on the Unified Modeling Language (UML'01), 2001, pp. 194-208

[18] Calamé, J., Ioustinova, N. and van de Pol, J., Automatic model-based generation of parameterized test cases using data abstraction. ENTCS, 191:25–48, 2007.

[19] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N. and Veanes, M., Model-based testing of ob ject-oriented reactive systems with spec explorer. Technical report, Microsoft Research, Redmond, May 2005.

[20] Chen, Y., Probert, R. L., and Ural, H. , Model-based regression test suite generation using dependence analysis. In Proceedings of the 3rd international Workshop on Advances in Model-Based Testing (London, United Kingdom, July 09 - 12, 2007). A-MOST '07. ACM, New York, NY, 54-62. DOI= http://doi.acm.org/10.1145/1291535.1291541

[21] Chen, Y., Rosenblum, D. S., and Vo, K., TestTube: a system for selective regression testing. In Proceedings of the 16th international Conference on Software Engineering (Sorrento, Italy, May 16 - 21, 1994). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 211-220.

[22] Clarke, E. M., Faeder, J., Langmead, C. J., Harris, L., Jha, S. K. and Legay, A., Distributed Statistical Model Checking of T Cell Receptor Pathway. In Proceedings of CMSB 2008.

[23] Du, W. and Mathur A. P., Testing for software vulnerability using environement perturbation. In Workshop on Dependability Versus Malicious Faults, International Conference on Dependable Systems and Networks (DNS2000), Proceedings, pages 603–612. IEEE Computer Society, 2000.

[24] Felderer, M., Breu, R., Chimiak-Opoka, J., Breu, M., Schupp, F., Concepts for Model-Based Requirements Testing of Service oriented systems. IASTED SE, 2009

[25] Felderer, M., Zech, P., Fiedler, F., Chimiak-Opoka, J., Breu, R., Model-driven System Testing of a Telephony Connector with Telling Test Stories. CONQUEST 2009

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

[27] Ferrante, J., Ottenstein, K. J., and Warren J.D., The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–49, July 1987.

[28] Fraser, G. and Wotawa F., Using Model-Checkers for Mutation-Based Test-Case Generation, Coverage Analysis and Specification Analysis. In Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), pages 16-22, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2703-5.

[29] Fraser, G. and Wotawa, F., Property Relevant Software Testing with Model-Checkers. SIGSOFT Softw. Eng. Notes, 31 (6): 1-10, 2006. ISSN 0163-5948.

[30] Fraser, G., Aichernig, K., Wotawa, F., Handling Model Change: Regression Testing and Test-Suite Update with Model-Checkers, vol. 190 of ENCTS, pp. 33-46, 2007. Springer.

[31] Frantzen, L., Tretmans, J. and Willemse T.A., Test generation based on symbolic specifications. pages 1–15. 2005.

[32] Gargantini, A. and Heitmeyer C., Using Model Checking to Generate Tests From Requirements Specifications. In ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 146–162, vol. 1687 of LNCS, 1999. Springer.

[33] IEEE, “Standard Glossary of Software Engineering Terminology,” IEEE Std. 610.12-1990.

[34] IEEE, “Software Engineering Body of Knowledge (SWEBOK),” ISBN 0-7695-2330-7, 2004.

[35] Jard C. and Jéron T., TGV: theory, principles and algorithms. Software Tools for Technology Transfer, 7(1):297–315, 2005.

[36] Jeffrey, D., Gupta, N., "Test suite reduction with selective redundancy," Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on , vol., no., pp. 549-558, 26-29 Sept. 2005

[37] Julliand, J., Masson, P.-A. and Tissot R., Generating security tests in addition to functional tests. In AST’08, pages 41–44. ACM Press, 2008.

[38] Julliand, J., Masson, P.-A. and Tissot R., Generating tests from B specifications and test purposes. In ABZ’08, volume 5328 of LNCS, pages 139–152, 2008.

[39] Jürjens, J., Secure Systems Development with UML, Springer-Verlag, 2005.

[40] Jürjens, J., Model-based Security Testing using UMLsec. Model-based Testing 2008 (co-located with ETAPS 2008).

[41] Korel, M., Tahat, L., Vaysburg, B., “Model Based Regression Test Reduction Using Dependence Analysis.” In Proceedings of the international Conference on Software Maintenance (Icsm'02) (October 03 - 06, 2002). ICSM. IEEE Computer Society, Washington, DC, 214.

[42] Leavens G. and Cheon, Y., Design by contract with jml. http ://jmlspec.org/jmldbc.pdf, 2006.

[43] Ledru, Y., du Bousquet L., Maur, O. and Bontron, P., Filtering TOBIAS combinatorial test suites. In FASE’04, volume 2984 of LNCS, pages 281–294, 2004.

[44] Le Guen, H., Marie, R., and Thelin T., Reliability estimation for statistical usage testing using Markov Chains. In ISSRE 04, St-Malo, Nov. 2004.

[45] Marchand, H., Dubreil, J. and Jéron, T., Automatic test generation for security property. Delivrable, Politess Project, 2008.

[46] Masson, P.-A., Julliand, J., Plessis, J.-C., Jaffuel, E. and Debois G., Automatic Generation of Model Based Tests for a Class of Security Properties. In A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing, London, UK, pages 12--22, July 2007. ACM Press.

[47] Mouelhi, T., Le Traon, Y. and Baudry B., Testing security policies: going beyond functional testing. In ISSRE’07 (Int. Symposium on Software Reliability Engineering), Trollhttan, Sweden, 2007.

[48] Myers G. J., The Arts of Software Testing. John Wiley & Sons, 1979 ISBN 0471469122

[49] Offutt, J.,Practical Mutation Testing,. Twelfth International Conference on Testing Computer Software, pages 99-109, Washington, DC, June 1995.

[50] Object Management Group (OMG), Model Driven Architecture, see www.omg.org/mda

[51] Pretschner, A. and Philipps, J., Methodological issues in model-based testing. In Model-Based Testing of Reactive Systems, pages 281–291, 2004.

[52] Rothermel, G. and Harrold, M.J., “Analyzing Regression Test Selection Techniques,” IEEE Trans. Software Eng., vol. 22, no. 8, pp. 529-551, Aug. 1996.

[53] Rothermel, J., Untch, R. H., Chu, C., Harrold, M.J., "Prioritizing Test Cases For Regression Testing," IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929-948, October, 2001.

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

[55] Schieferdecker, I., Modellbasiertes Testen, Objektspektrum, March 2007 (in German)

[56] Srivastava, A. and Thiagarajan, J. Effectively prioritizing tests in development environment. SIGSOFT Softw. Eng. Notes 27, 4 (Jul. 2002), 97-106. DOI= http://doi.acm.org/10.1145/566171.566187

[57] Tretmans, G.J., Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, volume 1055 of Lecture Notes in Computer Science, pages 127–146, Berlin, 1996. Springer Verlag.

[58] Utting M. and Legeard, B., Practical Model-Based Testing - A tools approach. Elsevier Science, 2006. Note: 550 pages, ISBN 0-12-372501-1.

[59] Vieira, M.E., Dias, M.S., Richardson, D.J., Object-Oriented Specification-Based Testing Using UML State-chart Diagrams, Proceedings of the Workshop on Automated Program Analysis, Testing, and Verification (at ICSE'00), June 2000

[60] Warmer, J. and Kleppe, A., The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison-Wesley, 2003

[61] Wong, W. E., Horgan, J. R., London, S., and Bellcore, H. A. A Study of Effective Regression Testing in Practice. In Proceedings of the Eighth international Symposium on Software Reliability Engineering (November 02 - 05, 1997). ISSRE. IEEE Computer Society, Washington, DC, 264.

[62] Xu, L., Dias, M. and Richardson, D. Generating regression tests via model checking. In COMPSAC'04, 28th Int. Computer Software and Applications Conference, pp. 336–341, 2004.

[63] Yang, Q., Li, J. J. and Weiss, D. M., "A survey of coverage-based testing tools," The Computer Journal, pp. bxm021+, May 2007. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxm021

[64] Zander, J., Dai, Z.R., Schieferdecker, I., Din G., From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. TestCom 2005

8 Tools

[65] Armando, A., Mantovani J. and Platania L., Bounded model checking of software using SMT solvers instead of SAT solvers. International Journal on Software Tools for Technology Transfer (STTT) 11(1), February, 2009

[66] Bardin S. and Herrmann P., Structural Testing of Executables. In Proc. First Int. Conf. Software Testing, Verification and Validation (ICST 2008), Lillehammer, Norway, April 2008.

[67] Benveniste, A., Le Guernic, P., Jacquemot, C. Synchronous programming with events and relations: the Signal language and its semantics. In Science of Computer Programming, v. 16, 1991.

[68] Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M. and Torreborre E., Model-based Testing from UML Models. In MBT'2006, Model-based Testing Workshop, INFORMATIK'06, volume P-94 of LNI, Lecture Notes in Informatics, Dresden, Germany, pages 223--230, October 2006.

[69] Blanc, B. and Marre, B.,. Test Selection Strategies for Lustre Descriptions in GATeL. MBT2004

[70] Cavalli, A., Montes De Oca, E., Mallouli, W., Lallali, M.. Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints, The 12th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2008).

[71] Chen, Y., Rosenblum, D. S., and Vo, K. 1994. TestTube: a system for selective regression testing. In Proceedings of the 16th international Conference on Software Engineering (Sorrento, Italy, May 16 - 21, 1994). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 211-220.

[72] Delmas, D. and Souyris J., ASTREE: from Research to Industry. Proc. 14th International Static Analysis Symposium, 2007, LNCS 4634, pp. 437-451.

[73] Dutertre, B. and Sorea, M., Timed Systems in SAL, Technical Report SRI-SDL-04-03, July 2004

[74] Franzle M. and Herde C., HySAT: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design, 30(3), pp179 198, June 2007

[75] Gallio Automation Platform. URL: http://www.gallio.org/Default.aspx

[76] Gaston, C. et al. Symbolic Execution Techniques for Test Purpose Definition. TestCom2006

[77] Garavel, H., Lang, F., Mateescu, R. and Serwe, W., CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes proceedings of the 19th International Conference on Computer Aided Verification CAV 2007, July 2007

[78] Gotlieb, A., Inka: An automatic software test data generator. In Proceedings of DAta Systems In Aerospace (DASIA 2001), Eurospace, The Association of European Space Industry, Nice, France, May 2001.

[79] Gotlieb, A., Euclide: A constraint-based testing platform for critical C programs. In 2th International Conference on Software Testing, Validation and Verification (ICST'09), Denver, CO, Apr. 2009.

[80] Goubault, E., Martel M. and Putot, S., Asserting the Precision of Floating-Point Computations: a Simple Abstract Interpreter. European Symposium on Programming, ESOP'02, LNCS2305, 2002

[81] Goubault, E. and Haucourt, E., A practical application of geometric semantics to the static analysis of concurrent programs. Proceedings of CONCUR 2005 in LNCS vol. 3653, Springer, 2005

[82] Hartman, A. and Nagin, K., The AGEDIS tools for model based testing, ACM SIGSOFT Software Engineering Notes, Volume 29 , Issue 4, July 2004.

[83] Henzinger, T.A., Ho, P.H., and Wong-Toi, H., HyTech: a model checker for hybrid systems. Journal of Software Tools for Technology Transfer, 1(1/2):110-122, 1997

[84] Holzmann, G. J., Design and Verification of Computer Protocols. Prentice Hall Int., 1991

[85] IBM. Rational Quality Manager, URL: https://jazz.net/projects/rational-quality-manager/

[86] Jard, C., Jeron, T., TGV: theory, principles and algorithms, A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems, Software Tools for Technology Transfer (STTT), 6, Octobre 2004

[87] Jeannet, B., Jeron, T., Rusu, V., Zinovieva, E., Symbolic Test Selection based on Approximate Analysis, in 11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), LNCS 3440 , pp 349-364, Edinburgh (Scottland), April 2005.

[88] Larsen, K. G., Pettersson P. and Yi, W., Uppaal in a nutshell. Journal of Software Tools for Technology Transfer, 1 (1-2):134-152, October 1997

[89] Le Guernic, P., Talpin, J.-P., Le Lann J.-C., Polychrony for system design Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. World Scientific, August 2003. Polychrony, Website: http://www.irisa.fr/espresso/polychrony

[90] MaTeLo tools by All4Tec, Website: http://www.all4tec.net

[91] Parasoft. Continous Regression Testing, available online. URL: http://www.parasoft.com/jsp/technologies/technologies.jsp?navIdx=1&subMenu=0&itemId=359

[92] PRISM: Probabilistic Symbolic Model Checker, Website: http://www.prismmodelchecker.org

[93] Prowell, S. J., JUMBL: A Tool for Model-Based Statistical Testing, 36th Annual Hawaii International Conference on System Sciences, 2003

[94] Test Designer, Website http://www.smartesting.com

[95] The Eclipse Foundation. Eclipse Test & Performance Tools Platform Project, URL: http://www.eclipse.org/tptp

[96] Tretmans, J., Brinksma, E., Cote de Resyste Automated Model Based Testing in 3rd Workshop on Embedded Systems, 2002, pp 246-255

[97] Visser, W. et al. Model Checking Programs. ASE Journal 2003

[98] Vojdani, V. and SeidlH., Region Analysis for Race Detection. Submitted to 16th International Static Analysis Symposium, 2009.

[99] Williams N. et al. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. EDCC 2005

[100] Website: www.prover.com

[101] Website:http://www.tni-software.com/en/produits/rtbuilder

[102] Website: http://www.systemc.org