Model-Based Testing for Functional and Security Test Generation

Fabrice Bouquet[1,2], Fabien Peureux[2], and Fabrice Ambert[2]

[1] Inria Nancy Grand Est – CASSIS Project
Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
fabrice.bouquet@inria.fr
[2] Institut FEMTO-ST – UMR CNRS 6174, University of Franche-Comté
16, route de Gray, 25030 Besançon, France
{fbouquet,fpeureux,fambert}@femto-st.fr

Abstract. With testing, a system is executed with a set of selected stimuli, and observed to determine whether its behavior conforms to the specification. Therefore, testing is a strategic activity at the heart of software quality assurance, and is today the principal validation activity in industrial context to increase the confidence in the quality of systems. This paper, summarizing the six hours lesson taught during the Summer School FOSAD'12, gives an overview of the test data selection techniques and provides a state-of-the-art about Model-Based approaches for security testing.

1 Testing and Software Engineering

One major issue, regarding the engineering in general and the software domain in particular, concerns the conformity of the realization in regards of the stakeholder specification. To tackle this issue, software engineering relies on two kinds of approaches: Validation and Verification, usually called *V&V*.

1.1 Software Engineering

The approaches proposed by Software engineering to ensure software conformity are the validation and the verification. There are many definitions of these two words but we propose to explain them in regards of the usage. The **validation** addresses the question *"Are we building the right product?"*, which aims to validate that the software should do what the end users really requires, i.e. that the developed software conforms to the requirements of its specification. The **verification** addresses the question *"Are we building the product right?"*, which aims to verify that all the artifacts defined during the development stages to produce the software conform to the requirements of its specification, i.e. that the requirements and design specifications has been correctly integrated in development stuff (model, code, etc.). It should be noted that another variants or interpretations of these definitions can be found in the literature, mainly depending of the engineering domain they are applied.

A. Aldini et al. (Eds.): FOSAD VII, LNCS 8604, pp. 1–33, 2014.
© Springer International Publishing Switzerland 2014

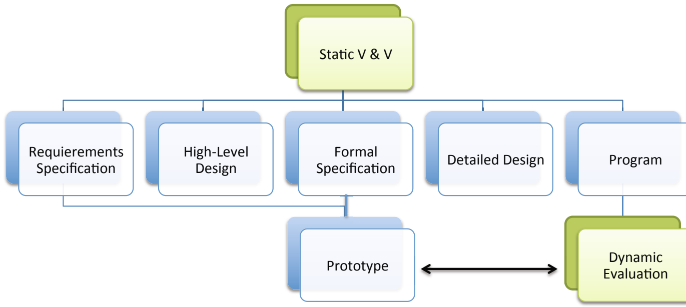


Fig. 1. Validation and verification

Figure 1 describes the different steps used by the methods of Verification and Validation. For each step, different techniques can be used:

- **Static Test** for the reviews of code, of specification, of design documentation.
- **Dynamic Test** for the execution of program to ensure the correctness of its functionalities.
- **Symbolic Verification** for run-time checking, symbolic execution (of model or code).
- **Formal Verification** for proof, model-checking from formal model.

The rest of this presentation focuses on dynamic test approach, which is today the principal validation activity in industrial context to increase the confidence in the quality of software.

1.2 What Is Testing?

Naturally, the next question is *"what is testing?"*. Since the first and the second techniques proposed in the previous part concern the Static Test and Dynamic Test, we propose three definitions of testing from the state of the art:

- IEEE Std 829 [1]: "The process of analyzing (execution or evaluation) a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item."
- G.J. Myers (The Art of Software testing [2]): "Test is the process of executing a program (or part of a program) with the intention of finding errors."
- E.W. Dijkstra (Notes on Structured Programming [3]): "Testing can reveal the presence of errors but never their absence."

Three major features can be underlined in these definitions. The first one concerns the capacity to compare the results, provided by the real system, to the expected value, defined in the specifications. It implies to have a referential to ensure the confidence in testing activity. The second one concerns the testing process itself, and the way to find bugs. Therefore, we need to define a coverage

strategy of the referential element to be tested (such as code, requirements or any artifact like models). Finally, the third issue is the uncompletion of testing because, in general, we cannot explore all possibilities. Indeed, due to combinatorial explosion of reachable states, exhaustive testing is unfeasible in practice, and dedicated strategies are needed to manage this explosion and to keep a relevant quality assessment.

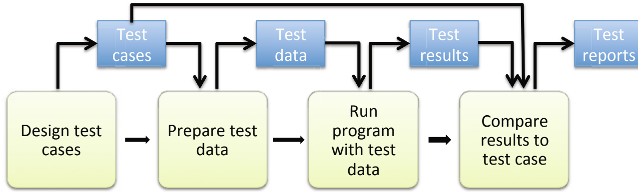


Fig. 2. Activities of dynamic test

As shown in Fig. 2, testing process can be decomposed in five main activities:

1. Select or design test cases: choose a subset of all possible features of the System Under Test (SUT). This information is provided by the validation and test plan. This step aims to produce the abstract test cases, i.e. scenarios that have still to be concretized using concrete values and function calls.
2. Identify the data for test cases: the test cases, defined in the previous step, can be used with several data. Then, it is important to choice a relevant subset of data. The number of test cases execution is equal to the number of chosen data.
3. Execute the test cases with the data: the test cases with data are executed on the SUT and test results are gathered using manual or automatic execution environment.
4. Analyze the results of the executed test. This step consists to assign a verdict to each executed test case by deciding if the test is in success or not:
   - Pass: the obtained results conform to the expected values.
   - Fail: the obtained results do not conform to the expected values.
   - Inconclusive: it is not possible to conclude.
5. Reporting: it is the evaluation of the quality and the relevance of the tests (to determine the need and effort to correct discovered bugs, to decide to stop testing phase, etc.).

Beyond this theoretical point of view, regarding the reality and practices of the industrial development context, the maturity of the Quality Assurance or Test function shift from theoretical ad hoc process to a more strategic and centralized approach. Moreover, the testing activity often concerns areas of IT organizations from western Europe Enterprise, which mainly promote and apply the following process steps[1]:

[1] Source IDC - European Services, Enterprise Application Testing Survey, March 2011.

1. Choose a testing methodology to address **agile/component** based development life cycle.
2. Provide **automated test coverage** that makes it possible to apply agility in testing.
3. Focus on the **non-functional aspects** like performance, availability, security, etc.
4. Refine the test strategy to **optimize the use of testing services** (traditional and cloud based).

This practical approach allows to highlight relevant aspects about testing activity. The first one is the acceptance of the need to test. In fact, the question is no more to know if testing should be used or not in the development process, but how to enhance its usage. Indeed, the integration of testing activity at the heart of the development process makes it possible to reduce bottleneck of the project deadline and avoid the crushing tension of the validation phase before releasing the product. Therefore, optimization issues are studied and generalization are discussed to generalize it for other aspects (up to unit and/or functional for example). The automation of the testing activity and the improvement of the confidence level are thus today the main issues about testing.

1.3 Kinds of Test

"Testing" is a very generic term. In reality, as suggested in previous IDC survey, thus we need to refine it. J. Tretmans in [4] proposed to decompose the testing approaches regarding three dimensions. This representation is shown in Fig. 3.

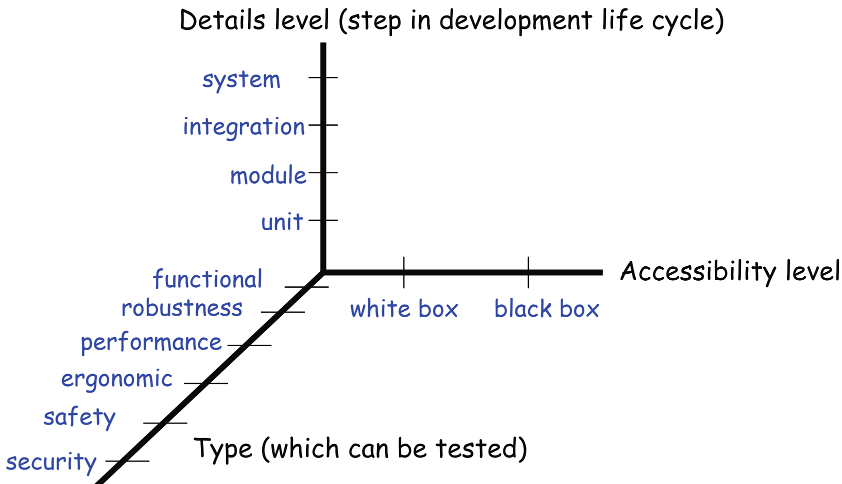


Fig. 3. Kind of testing approaches as proposed by J. Tretmans

The first dimension (horizontal axis) concerns the accessibility of the code to be tested. *White box* defines an open box in which all the information about the code are accessible. *Black box* defines a closed box and only the binaries are available to execute the test. The second dimension (vertical axis) is the architecture level of the system to be tested, i.e. the development phase from the software life cycle point of view. An example of the development life cycle and validation steps is proposed in Fig. 4. The figure depicts four phases from the specification, provided by the client, to the code implementation. Each described phase has four specific level or context, which are addressed by dedicated test objectives. For example, Unit testing focuses on the coverage of the code to separately validate the computation of each implemented functions or procedures, while Acceptance testing focuses on the end-user features and thus addresses only the functional (or domain) requirements.

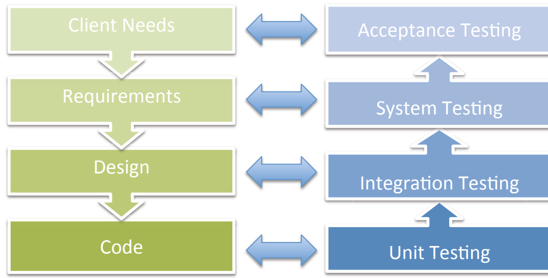


Fig. 4. Development life cycle & testing levels

Finally, the third dimension (plan axis) is the usage or targeted aspects to be validated by the tests. So, each test is an element that can be located in this 3D space representation.

White Box. White box testing, also called Structural Testing, instantiates the four activities of the Fig. 2 as follows:

- Test cases: they cover all the functions of the source code to be tested.
  - Deriving from the internal design of the program.
  - Requiring detailed knowledge of its structure.
- Data: test data are produced from the source code analysis in order to ensure some given coverage criteria. The most common coverage criteria are based on the analysis and coverage of the flow graph associated to a function of the source code as illustrated in Fig. 5.
  - Statement: node or block of instructions,
  - Branch: condition,

    - Path: a problem regarding the path coverage is the number of loops (for example between B and A in Fig. 5), so there exist more restrictive criteria as *independent path* or *k-path* (where k defines the maximum number of loops to be tested). A comparison and a hierarchy of all such structural criteria are proposed in [5].
- Execution: the test scripts take the form of a suite of function calls (with data) of the source code.
- Reporting and end of the testing phase: it directly depends on the completion of the given targeted coverage criteria.

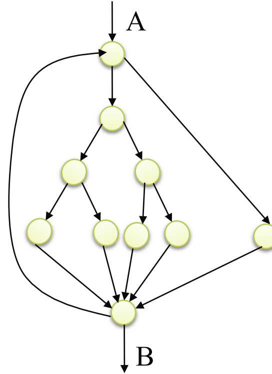


Fig. 5. Source code representation for structural testing

Black Box. Black box testing, also called Functional Testing, instantiates the four activities of the fig. 2 as follows (the overall black box testing process is also depicted in Fig. 6):

- Test cases are derived from the functional specification.
    - Designed without knowledge of the source code internal structure and design.
    - Based only on functional requirements.
- Test data are also derived from the functional specification by applying dedicated test coverage criteria on the related specification in order to identify and target some requirements or functionalities as test objectives.
- Execution: the test scripts take the form of a suite of API call or implemented user actions (with concrete data) of user or software interfaces of the program.
- Reporting and end of the testing phase: it directly depends on the completion of the given targeted coverage criteria.

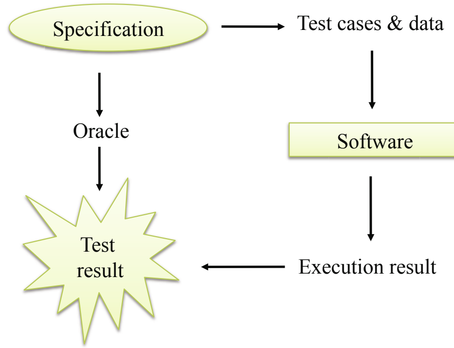


Fig. 6. Functional testing process

1.4 Testing Universe

In fact, testing is in the middle of many artifacts, called test universe. For instance, the specification explains, in term of needs and requirements, the features that have to be developed, and also what the tests have to validate. It also identifies the defects to be avoided (from which associated test can be derived). A same test can be used to validate that a requirement is correctly implemented (at its first execution), but it can also be used to ensure this requirement remains correctly implemented in future versions (this kind of test is called *non regression test*). It is also possible to create or derive test from specific artifacts such as contracts or models, which provide an abstraction level to help and ease the design of the tests. Finally, as depicted in Fig. 7, some tooling and environment are now available to manage, in an automated manner, all these artifacts related to requirements, models, defects, test scripts and test definition repository. Those make it possible to offer a scalable and reliable automation of the testing process, and ease its usage in the industrial development and validation team.

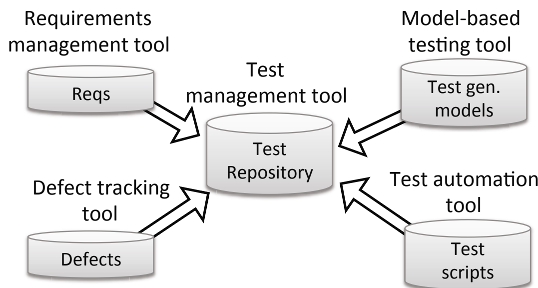


Fig. 7. Sphere of testing tools

To conclude this overview, testing is today in industry the main activity to determine if a developed software does the right things and things right (using each level of test). It represents until 60% of the complete effort for software development. This effort is reported in 1/3 during software development and 2/3 during software maintenance. Nevertheless, in current software development, testing is still often seen as a necessary evil. It indeed remains a not very popular activity in development teams especially because integrating testing activities during development phase give rise to a psychological hindrance and/or cultural brake because testing is seen as a destructive process (an efficient test is a test that reveals an error), whereas programming is seen as a constructive and helpful process (each line of code aims to fulfill a functional need for the end users). Moreover, teams given the task of testing are often appointed by default and have no professional skills in testing activities, although specific knowledge and competences are required for the task. This paper precisely aims to fill in the gaps on testing knowledge by introducing practical testing approaches focusing on functional testing and security test design.

The paper is organized as follows. Techniques for test data selection are introduced in Section 2, functional Model-Based Testing approaches are described in Section 3, and specificities about Model-Based Testing to address security features are presented in Section 4. Finally, Section 5 concludes the paper and gives several tooling references to look at practical Model-Based Testing in more depth.

2 Test Data Selection

The previous section introduced the testing steps to design test cases. This section deals with the strategic activity concerning the selection of the data that will be used to build executable test cases. Three main techniques are used to choice the test data. The first one uses a partition analysis based on the input domains. This approach reduces the number of possible values by selecting a representative value (bound, middle, random...) for each identified partition. The second one consists to apply a combinatorial testing strategy. This approach can also restrict the possible combination of the input values by selecting specific n-uplets of values. The third one is based on random or stochastic testing. This approach enables to choose different input values using statistic strategies that allow an uniform distribution of the selected data.

2.1 Partition Analysis of Input Domains

This technique, based on the classes of equivalence, is control flow oriented. A class of equivalence corresponds to a set of data supposed to test the same behavior, i.e. to activate the same effect of the tested functionality. The definition of equivalent classes thus allows to convert an infinite number of input data into a finite number of test data. The computation of the test data are performed in order to cover each identified behaviors. For example, let a given function defined by:

Domain: $x \in -1000..1000$
Precondition: $x \leq 100$
Postcondition:
IF $x \leq 0$ **THEN** $y \leftarrow$ default
ELSE IF $x \leq 40$
  THEN $y \leftarrow$ low
  ELSE $y \leftarrow$ high
  END
END

The partition analysis of this function gives rise to identify three behaviors P_i in regards to Pre and Postcondition:

1. P_1: $x \leq 0$
2. P_2: $x > 0 \wedge x \leq 40$
3. P_3: $x > 40 \wedge x \leq 100$

We can then derive the corresponding constraints about the domain of the variable x and define four classes of equivalence C_i:

1. C_1: $x \in [-1000, 0]$
2. C_2: $x \in [1, 40]$
3. C_3: $x \in [41, 100]$
4. C_4: $x \in [101, 1000]$

This technique thus aims to select data in order to cover each behavior of the tested function. Such function often includes control points, which are usually (such as the previous example) represented by IF-THEN-ELSE construct in programming languages. The executed effects of a given execution depend on the evaluation of the boolean formula, called *decision*, which is expressed in the IF statement and gives rise to two equivalent classes : one makes true the evaluation of the decision, the other makes it false. In the previous example, this decision is an atomic expression ($x \leq 0$), but in practice decisions are complex predicates constructed with $\wedge$, $\vee$ and $\neg$ operators, combining elementary boolean expressions, called *conditions*, that cannot be divided into further boolean expressions. Exposing the internal structure of a decision can lead to extend the number of equivalence classes if each evaluation of each condition is considered. This issue of how to treat multiple conditions without exponential test case explosion is a key point for test generation. Several structural coverage criteria for decisions with multiple conditions have thus been defined in the testing literature. Brief informal definitions and hierarchy (see Fig. 8) are given here, but more details including formal definitions in Z are available elsewhere [6,7]. Note the terminology: a *decision* contains one or more primitive *conditions*, combined by disjunction, conjunction and negation operators.

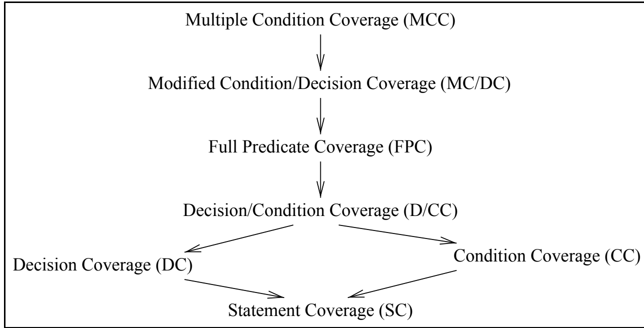


Fig. 8. The Hierarchy of control-flow coverage criteria for multiple conditions. $C_1 \longrightarrow C_2$ means that criterion C_1 is stronger than criterion C_2.

Statement Coverage (SC). The test set must execute every reachable statement of the program.

Condition Coverage (CC). A test set achieves CC when each condition in the program is tested with a true result, and also with a false result. For a decision containing N conditions, two tests can be sufficient to achieve CC (one test with all conditions true, one with them all false), but dependencies between the conditions typically require several more tests.

Decision/Condition Coverage (D/CC). A test set achieves D/CC when it achieves both decision coverage (DC) and CC.

Full Predicate Coverage (FPC). A test set achieves FPC when each condition in the program is forced to true and to false, in a scenario where that condition is *directly correlated* with the outcome of the decision. A condition c is directly correlated with its decision d when either $d \iff c$ holds, or $d \iff \neg c$ holds [8]. For a decision containing N conditions, a maximum of $2N$ tests are required to achieve FPC.

Modified Condition/Decision Coverage (MC/DC). This 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 decision's outcome by varying just that condition while holding fixed all other possible conditions [9,10]. Achieving MC/DC may require more tests than FPC, but the number of tests generated is generally linear in the number of conditions.

Multiple Condition Coverage (MCC). A test set achieves MCC if it exercises all possible combinations of condition outcomes in each decision. This requires up to 2^N tests for a decision with N conditions, so is practical only for simple decisions.

These different coverage criteria can also be used to as data selection criteria by rewriting the decision into several predicates. Each predicate defines a specific equivalent class in which data have to be derived. To achieve it, a simplistic way is to consider a single disjunction $A \vee B$ nested somewhere inside a decision.

We propose four possible rewriting rules to transform the disjunction into a set of predicates defining data equivalent classes:

- $A \vee B \rightsquigarrow \{ A \vee B \}$. This generates just one equivalent class for the whole disjunct, resulting in one test for the whole decision. This corresponds to decision coverage (because the negated decision is achieved by another equivalent class, e.g., corresponding to the ELSE branch).
- $A \vee B \rightsquigarrow \{ A, B \}$. This ensures D/CC, because there is one equivalent class defined by A true, and one by B true, and another one with the negated decision that will cover $\neg A \wedge \neg B$. In fact, a single equivalent class, $A \wedge B$, would in theory be enough to ensure D/CC, but $A \wedge B$ is often not satisfiable, so two weaker tests are generated instead.
- $A \vee B \rightsquigarrow \{ A \wedge \neg B, \neg A \wedge B \}$. This is similar to FPC, because the result of the true disjunct is directly correlated with the result of the whole disjunction, since it cannot be masked by the other disjunct becoming true.
- $A \vee B \rightsquigarrow \{ A \wedge \neg B, \neg A \wedge B, A \wedge B \}$. This corresponds to MCC, because it defines an specific equivalent class for each combinations of A and B (the $\neg A \wedge \neg B$ combination is covered by the negated decision). This usually becomes unmanageable even for moderate values of N conditions.

Figure 9 depicts these rewriting rules by showing the different equivalent classes: the regions A and B identify the data domain of the two conditions A and B of the decision $A \vee B$.

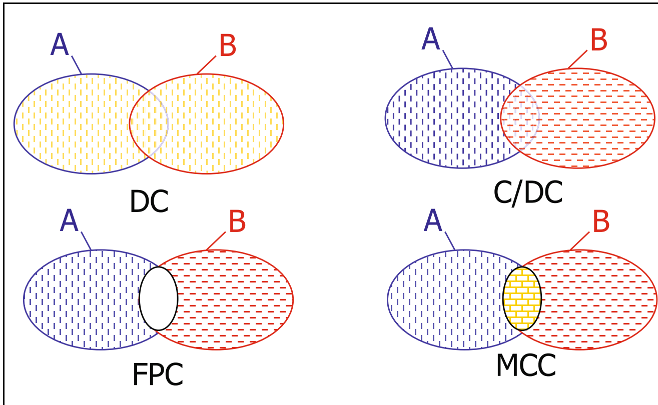


Fig. 9. Decision coverage

Finally, selecting data from obtained equivalent classes can be performed non-deterministically, but also by applying a boundary/domain approach [11]. This boundary values approach is known to be an efficient strategy to select test data and is currently used as the basis for test generation algorithms [12], but it has not generally been formalized as coverage criteria.

To achieve this approach, some simple rules, based on the type of the parameter, can be applied to choice the values of data. Basically, for each equivalent class, it consists to take the data value at an extremum – minimum or maximum – of its domain. It should be noted that this approach can only be performed if an evaluation function can discriminate each value of the domain (minimum or maximum of integers, minimum or maximum of the cardinality of sets, etc.), else an arbitrary value can be selected by default as usual. Some examples illustrating this approach are given below:

- for each interval of integer described by an equivalent class, we select 2 values corresponding to the extrema (minimum and maximum), and 4 values corresponding to the values of the extrema with minus/plus delta:
  $n \in 3..15 \Rightarrow v1 = 3, v2 = 15, v3 = 2, v4 = 4, v5 = 14, v6 = 16$
- if the variable takes its value in an ordered set of values, we select the first, the second, before the last and the last data and one data outside the definition set:
  $n \in \{-7, 2, 3, 157, 200\} \Rightarrow v1 = -7, v2 = 2, v3 = 157, v4 = 200, v5 = 300$
- for the data defining an object, we can minimize or maximize some feature of its format, by selecting valid extremum values
- for an input file containing 1 to 255 records, we can select files with: 0, 1, 255 and 256 records
- for an object p typed by a static type C:
  - `null` reference
  - `this` reference (if \typeof(this) <: \type(C))
  - One object such that: p != `null` && p != `this` && \typeof(p) == \type(c)
  - One object such that: p != `null` && p != `this` && \typeof(p) ∈ \type(c)
  - One object such that: p == p' with p' an other compatible object

When functions have several parameters (inputs), the global approach has to be applied for each parameter. The first step consists, for each input, in calculating their domain from equivalent classes. The second is to select representative value(s) of each domain. The third consists to use a composition (by Cartesian product as instance) of all selected input values to generate the test data. This composition can lead to a combinatorial explosion that need to be mastered to make the test data set manageable. The next section gives an overview of techniques to achieve that in the more global context of combinatorial testing.

2.2 Combinatorial Testing

The combination of all possible (or selected) input values can give rise to combinatorial explosion of the configurations. For example, from two inputs defined as integer, we obtain: $2^{32} * 2^{32} = 2^{64} = 18\ 000\ 000\ 000\ 000\ 000\ 000$ possible configurations. Another concrete example concerns the preference parameter to fix a character styles (see Fig. 10). The form is composed by seven check boxes and one pull-down menu with four entries: it thus defines $2^7 * 4 = 512$ data possible combinations.

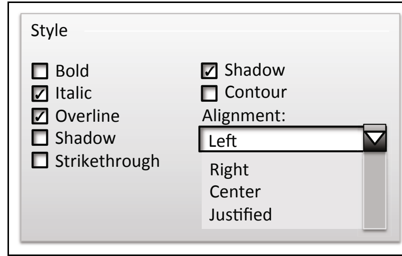


Fig. 10. GUI example for combinatorial testing

To control this combination, a classic approach is the Pair-Wise strategy. It aims to test a fragment of the value combinations such that they guarantee that each combination of two variables is tested. Indeed, practice shows that a majority of bugs can be detected by only covering the combinations of two data. For example, given the four following inputs representing:

- the operating System (OS): Windows, Mac Os, Linux,
- the Network connection: Cable, Wifi, Bluetooth,
- the file format: text, picture, mixed text picture,
- the printer technology: laser, liquid inkjet and Solid ink.

To cover all the possible configurations for the four inputs with a domain of 3 values, we must generate: $3^4 = 81$ test data. The Pair-Wise approach makes it possible to cover all the combinations of two values with only nine test data as shown in Tab. 1.

Table 1. Pair-Wise results

Case	OS	Network	Format	Printer
1	Windows	Bluetooth	laser	Text
2	Mac OS	Cable	Liquid	Text
3	Mac OS	Wifi	laser	Picture
4	Windows	Cable	Solid	Picture
5	Windows	Wifi	Liquid	Mixed
6	Linux	Bluetooth	Liquid	Picture
7	Linux	Cable	laser	Mixed
8	Mac OS	Bluetooth	Solid	Mixed
9	Linux	Wifi	Solid	Text

It is also possible to combine more values using a N-wise approach, in which N defines the number of data value to be associated (N=2 for Pair-wise, N=3 for Triplet-Wise, N=4 for Quadruplet-Wise, etc.). However, it should be noted that the number of test cases can quickly increase. More details (various articles and tools) about this kind of strategies can be found at the following website: `http://www.pairwise.org/default.html`.

2.3 Random and Stochastic Testing

This kind of testing approaches replace the partition analysis. Basically, their principles are to apply a random function to select the test data. The random function allows to take a value in the domain of the input in a nondeterministic way or using statistic laws [13,14,15]. For example, to select a data representing a distance, a sampling rate of 5 units to extract a set of input data to be tested; to choose data to represent the size of an individual, a law of Gauss can be applied.

The interest of such approaches concerns the simple way of automation to perform the test data selection, even if the expected result can be more difficult to predict. The objectivity of the test data is assumed by the blinding research. However, the blinding research could be a problem because it is difficult to generate real-life use case with arbitrary automated process.

The case studies show that the statistical testing approaches make it possible to quickly achieve 50% of the testing objective, but, as described in Fig. 11, it has a tendency to stagnate at this rate.

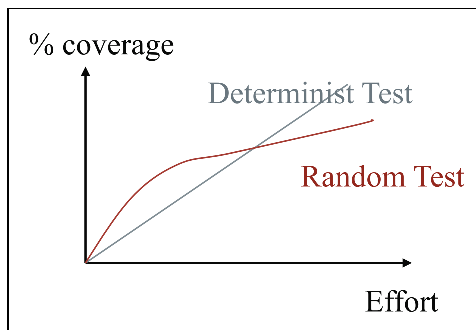


Fig. 11. Random testing achievement

3 Functional and Model-Based Testing

Functional testing aims to validate the system under test from a behavioral point of view, i.e. to ensure it conforms to the required functional requirements of its specification. It requires that the system under test is checked according to predetermined and expected behavior under specific circumstances. As shown in Fig. 3, this kind of test is therefore performed to an upper level since it involves the system as a product ready to be used (black box approach), and not pieces of code as structural testing does (white box approach). However, regarding test generation techniques, both domain are close since the strategies to select test data, introduced in previous Section 2, have been adapted to be applied to functional approaches [11,16]. Next subsection illustrates the application of the test data selection based on equivalent classes (introduced in Section 2.1) in the context of functional testing.

3.1 Example of Data Selection for Functional Testing

This use-case example deals with a secure register form of a web site. The goal of this use-case is to provide the test cases to validate this application. Figure 12 provides a simple version (V_1) of the form.

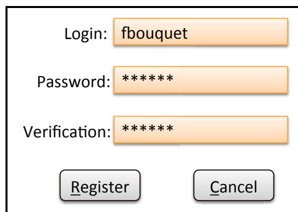
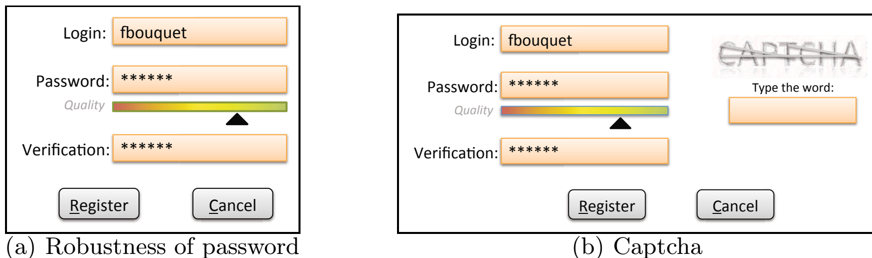


Fig. 12. GUI of the register form (version V_1)

From the interface of the Fig. 12, we can define five test objectives expanded (with data) into twelve test cases:

1. The `Login` field is kept empty or not. For non empty scenario, we can decide to fill with one character (minimal size), 8 characters (classical size) and 256 characters (huge size). Therefore, 4 different test data can be derived.
2. The `Login` exists in the system or not (2 test data).
3. The `Password` field is kept empty or not (2 test data).
4. The `Password` and `Verification` (to enter again the password) fields are the same or not (2 test data).
5. The security aspect is assumed by the protocol to be used: HTTP or HTTPS (2 test data).

Possible extensions of the form can also be handled by considering a greater level of robustness regarding password security (as proposed in version V_1 of Fig. 13(a)), and by increasing the protection against robot using captcha technique (as proposed in version V_2 of Fig. 13(b)).



(a) Robustness of password (b) Captcha

Fig. 13. GUI of the register form (version V_1 and V_2)

To derive test cases from the aggregating version V_2, we can reuse the 12 test data of the previous version, and complete them using two additional test objectives that can be expanded into 5 test data (so 17 test data in all):

– Verification of the quality / robustness of the entered password (3 data - one by level: poor, average, good).
– Verification of the captcha word by succeeding or not the challenge (2 data).

The equivalent class approach to select test data is therefore an efficient way to maximize the functional coverage of a test suite with a minimal set of data. The major weakness of this approach concerns the lack of automation: test data are manually designed and the expected results have to be empirically checked. This lack of automation makes repeated and tedious the activity of test case design and verdict assignment. To overcome this problem, Model-based Testing provides an automated approach by using a formal test model to derive test cases, predict the test results, and compare obtained results with expected ones to assign the verdict [17]. The automation of such test generation process is a strategic issue, since it can replace the (so current) manual development of test cases, which is known as costly and error-prone [18]. The next subsection introduces this functional testing approach.

3.2 Model-Based Testing Overview

Model-based testing (MBT) is an increasingly widely-used technique, relying on (semi) formal models called test models, for automating the generation of tests [19]. There are several reasons for the growing interest in MBT approach:

– The complexity of software applications continues to increase, and the user aversion to software defects is greater than ever, so the testing process has to become more and 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), so there is a strong push to investigate methods like MBT that can decrease the overall cost of test by designing tests automatically as well as executing them automatically.
– The MBT approach and the associated tools are now mature enough to be applied in many application areas, and empirical evidence is showing that it can give a good Return On Investment.

The main benefits of Model-Based Testing can be summarized as follows:

– It shortens the testing cycle by starting test automation before the application is available. The test models, and the derived test cases, can indeed be realized independently of the development progress.
– It enables to detects bugs sooner with the earlier involvement of testers in the development process (which can be seen as another cost-effective benefit).
– It reduces test execution costs since test cases can be concretized and executed automatically. Execution of automated tests can also be done overnight.
– It improves the overall quality of the test cases: test case generation is computed in an automated manner and is therefore more predictive and less error-prone than manual processes.

– It increases the scope and the value of regression testing: the test cases, based on the same model, can be generated for various implementations, releases, and versions of a single application, which ensures efficient regression testing.
– It reduces test maintenance effort since the test model becomes the single reference source of the testing process, and it is usually easier to manage this model rather than to directly update the test cases (it is a key item when features change constantly).

In this way, MBT approach renews the whole process of software testing from business requirements to the test repository, with manual or automated test execution by supporting the phases of designing and generating tests, documenting the test repository, producing and maintaining a bidirectional traceability matrix between tests and requirements, automating test verdict assignment and finally accelerating test automation [20]. The global picture of the MBT process is shown in Figure 14. The first step of this approach consists to specify a test model that captures the functional behavior of the system under test. From this model, test cases can be automatically computed using algorithms or designed manually to feed a test repository. The computed test cases are often abstract because they are defined at the same abstraction level than the test model: a dedicated publisher makes it possible to produce, from the abstract test cases, executable test scripts. Afterwards, executable scripts or informal scenarios can be executed on the concrete system to be tested. The test results and verdicts can then be saved to be used as test report. It should be noted that some artifacts of this process have already been introduced in Fig. 7 regarding test universe.

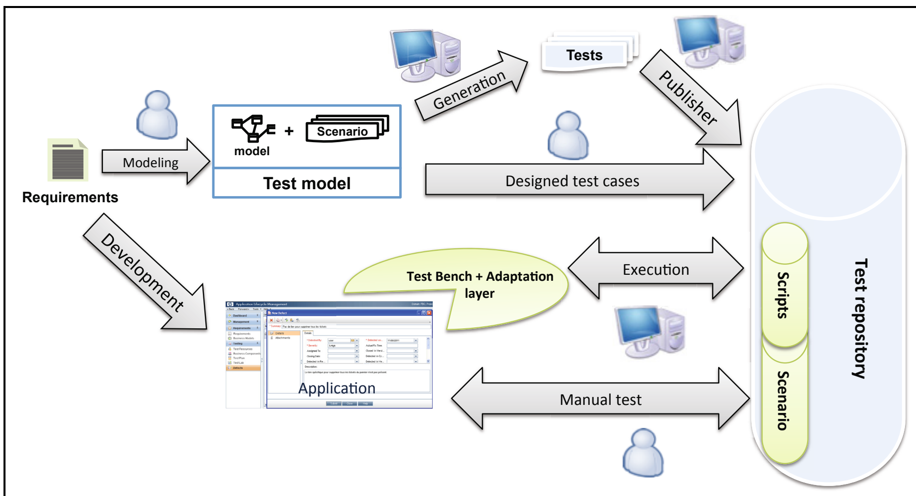


Fig. 14. Model-Based Testing architecture

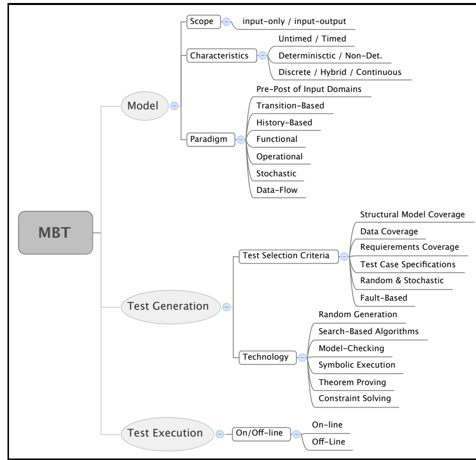


Fig. 15. Model-Based Testing taxonomy

Many approaches and techniques can be used to apply MBT process. In [21], the authors propose a taxonomy based on the modeling, test generation and execution paradigm. Figure 15 summarizes each of the paradigms identified by the authors to provide this taxonomy:

Model Scope. The test model can be based on the requirements associated to the inputs only or both to inputs and outputs.

Model Characteristics. The test model can capture some features of the system under test regarding temporal aspects, non determinism, events or continue values,...

Model Paradigm. It refers to all (semi-) formal model defined since time immemorial ... from the hieroglyph to SysML [22] including Z [23], B [24], Lustre [25], etc.

Test Generation Criteria. It concerns the coverage criteria used to drive the test generation process, and which should be ensured by the generated test cases. Sometimes, it could be a budget coverage criterion: the industry answer to *"when is testing done?"* can be understanding by *"when there is no more money"* or *"when the deadline is reached"* (however, the answer often relies on a rational approach!). Since adequacy criteria can lead to answer the wrong response, the test criteria selection is a crucial choice in regard to constraints (time or resources). That is why a practical evaluation and comparison of approaches must be considered to make the right choice.

Test Generation Technology. It relates to the interpretative semantics associated to the test model in order to automatically derive the test cases.

Test Execution. It concerns the interactions between the test execution process and the system under test. The test cases can be directly executed during the generation process (on-line approach) or not (off-line approach). Using on-line approaches makes it possible to interact with the system under test, and to dynamically use its outputs to adapt the test generation algorithms (to choose the inputs of the next stimuli as instance).

3.3 Example of MBT Approach for Functional Testing

This section illustrates a such MBT approach, based on a UML test model, using a simple example of Web application, namely eCinema. Basically, eCinema is a simple web-application that allows a customer to buy tickets on line before to go to his favorite cinema. The main screen of the application displays the list of available movies and show times. Before selecting tickets, a user should be logged to the system. This requires a registration. A registration is valid when a user gives a name (not already used) and a valid password. A valid new registration implies that the user is automatically logged in. When logged in, the user can buy tickets. If there are available tickets he can see his basket to verify his selection. When checking his selection, the user can delete tickets and then the number of available tickets for the session is automatically updated. The functional requirements of the application are described in Table 2.

Table 2. Requirements of eCinema website example

#	Requirements	Description
1	ACCOUNT_MNGT/LOG	The system must be able to manage the login process and allow only registered user to login.
2	ACCOUNT_MNGT/ REGISTRATION	The system must be able to manage the user's accounts.
3	BASKET_MNGT/ BUY_TICKETS	The system must be able to allow users to buy available tickets.
4	BASKET_MNGT/ DISPLAY_BASKET DISPLAY_BASKET_PRICE	The system must be able to display booked tickets and the total basket's price for a connected user.
5	BASKET_MNGT/ REMOVE_TICKETS	The system must be able to allow deletion of all tickets for a given user.
6	CLOSE_APPLICATION	The system can be shut down.
7	NAVIGATION	It is possible to navigate from one state to another.

The requirements are translated into a UML test model written with a subset of UML/OCL (called UML4MBT [26]). Concretely, a UML4MBT model consists of (i) UML class diagrams to represent the static view of the system (with classes, associations, enumerations, class attributes and operations), (ii) UML Object diagrams to list the concrete objects used to compute test cases and to define the initial state of the SUT, and (iii) state diagrams (annotated with OCL constraints) to specify the dynamic view of the SUT.

Figures 16, 17 and 18 respectively show the UML class diagram, Object diagram and state diagrams with an excerpt of OCL constraints that describe the eCinema example.

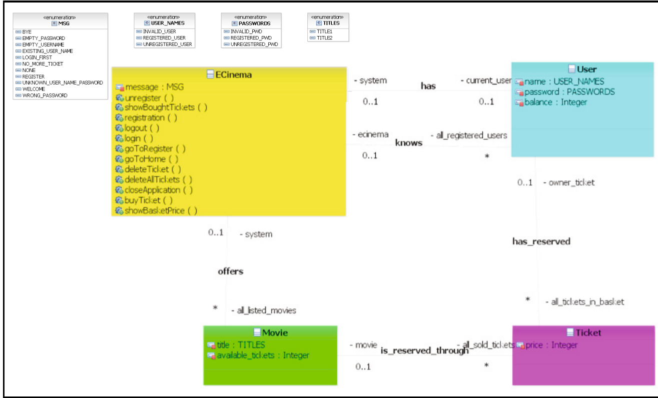


Fig. 16. eCinema class diagram

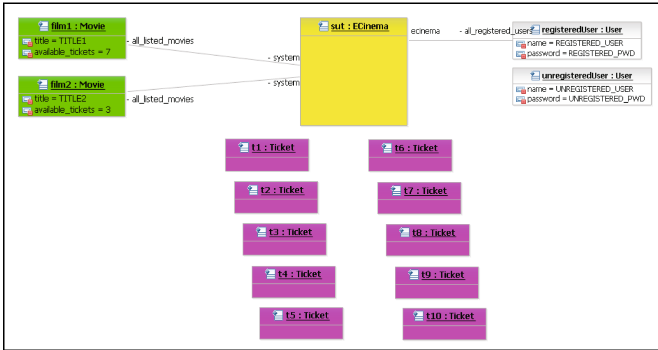


Fig. 17. eCinema object diagram

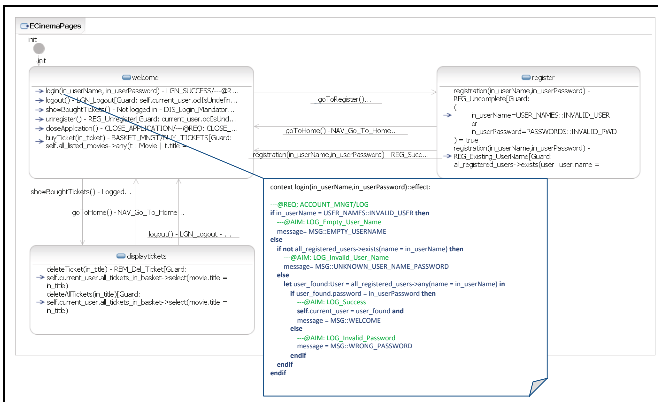


Fig. 18. eCinema state diagram with OCL constraints

These three diagrams enable to simulate the execution of the eCinema application and to automatically generate test cases by applying predefined coverage strategies (such as D/CC) on OCL constraints. The generated test cases and expected outputs are then published into a test repository, namely Testlink[2], as depicted in Fig. 19. During this step, a manually-designed mapping table concretizes the abstract generated test cases into executable scripts by translating the UML data into concrete ones. More details about this MBT testing approach can be found in [27].

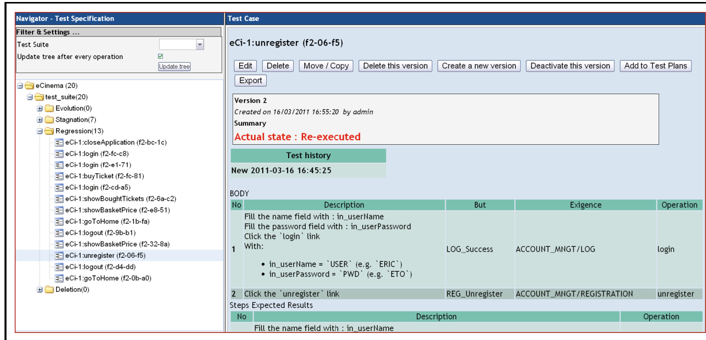


Fig. 19. Management of generated test cases using Testlink

The next section introduces the features of security testing and shows how such MBT processes can be efficiently used for this specific testing domain.

4 MBT Approach within Security Testing

Software security testing aims at validating and verifying that a software system meets its security requirements [28]. It targets two principal testing domain: functional security testing and security vulnerability testing [29]. Functional security testing is used to check the functionality, efficiency and availability of the designed security functionalities and/or security systems (e.g. firewalls, authentication and authorization subsystems, access control). Security vulnerability testing (or penetration testing, often called pentesting) directly addresses the identification and discovery of system vulnerabilities, which are introduced by security design flaws or by software defects, using simulation of attacks and other kinds of penetration attempts.

The security testing techniques can be divided into four families as shown in Fig. 20. The first one is the network security toolkit, with the network scanners to check active ports and (characteristics of) computers on the network. The second one concerns the Static Application Security Testing (SAST), which aims to analyse application regarding known security threats using tools such as

[2] http://testlink.org/

HP/Fortify, Veracode, Checkmarx, Parasoft... The third family focuses on monitoring approach, which consists to capture and analyse the behaviors and events on the network using tools like Syslog, Nagios or IBM Tivoli... Finally, the fourth family relates to Dynamic Application Security Testing (DAST) that consists to dynamically check the security requirements. Typically, DAST techniques can be performed using model-based testing approach dedicated to security features.

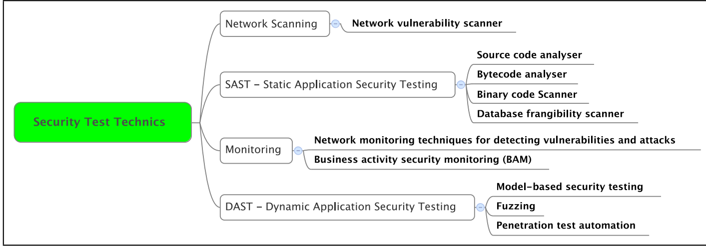


Fig. 20. Taxonomy of security software testing

In fact, recent IBM X-Force© research revealed that, in 2012, 41% of all security vulnerabilities pertained to web applications as shown in Fig. 21. This kind of attacks is more and more complex and can be usually discovered only using a dynamic approach. The rest of this section thus deals with on DAST techniques, with a specific focus on MBT security testing approaches.

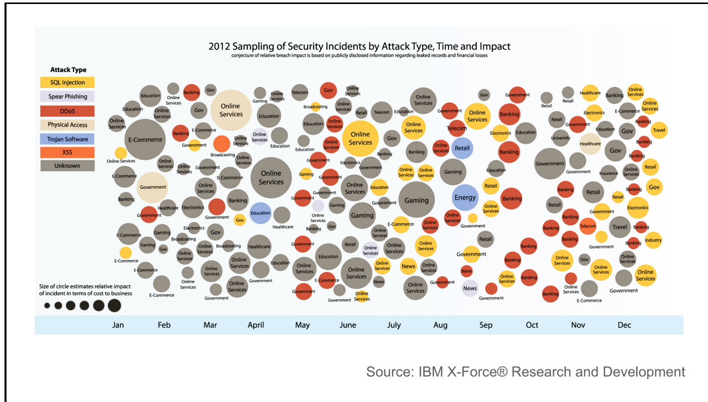


Fig. 21. Attack evolution from IBM X-Force

For further details about Web application security attacks, the Open Web Application Security Project (OWASP) proposes some documentation including a current Top Ten of the current threats [30]. The most prevalent and dangerous cyber-attacks against Web Applications are also reported and available in CWE/SANS 25 [31] and WhiteHat Website Security Statistic Report 2013 [32].

4.1 Model for Security Testing

M. Felderer et al. propose in [33] a classification for Model-Based Security Testing decomposed into five families:

Individual Knowledge. The individual knowledge determines the design of security tests. It is also used to select function and data to be tested.

(Adapted) Risk-Based Testing. These techniques are based on threat models and enable the prioritization of test concept or execution.

Scenario-Based MBT. It concerns techniques to complete test models (of MBT approach) using scenarios dedicated to security aspects.

Risk Enhanced Scenario-Based MBT. This kind of approaches completes the (MBT) test models using risk information in addition to the scenarios.

Adapted MBT. It relates to all other MBT approaches that use a dedicated test model for security.

In many families presented for security testing, the link between risk and testing is very important. As described in Fig. 22, risk assessment activity can drive the MBT approach. In fact, each step of the MBT approach (modeling, criteria to drive test generation and prioritization of the test execution) are driven by the results of risk analysis and assessment. One of the more mature approach addressing Model-Driven Risk Analysis is CORAS [34], which provides a customized language for threat and risk modeling. More precisely, CORAS is a model-driven method for risk analysis featuring a tool-supported modelling language specially designed to model risks that are common for a large number of systems. Such model serves as a basis to perform risk identification and prioritization.

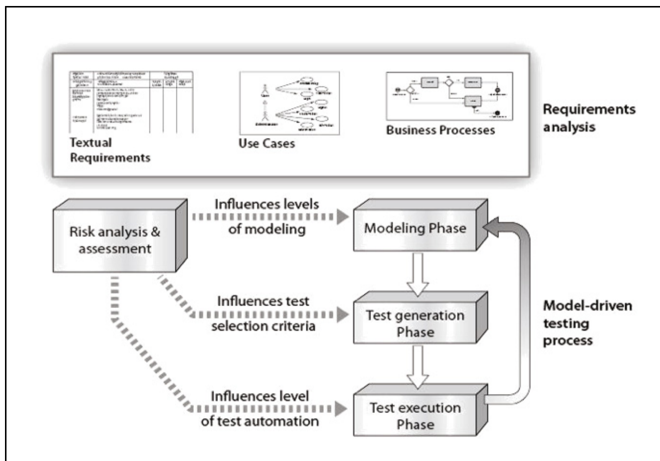


Fig. 22. Link between MBT and risk assessment

4.2 Security Test Objectives

To define security test objectives, two main approaches have been defined during the last decade. The first approach is based on dedicated security test models as proposed in [35,36] with UMLSec, or as previously proposed with SecureUML in [37]. These approaches can have a specific focus on protocols as proposed in [38] using finite state machine, or using networks rules [39], or using protocol mutation as described in [40]. In parallel, some other techniques are emerging to help analysis like Threat or Risk model like CORAS [34].

We can also find specific testing techniques such as Fuzz testing (or Fuzzing) as proposed in [41], which was used by Microsoft company to validate the layer that manages the data and files acquired from network [42]. Fuzz testing, originated from B.Miller at the University of Wisconsin [43], involves providing invalid, unexpected, or random data to the inputs of a system under test. Although its origin is based on a complete randomized approach, more systematic approaches have been recently proposed: model-based fuzzers use their knowledge about the message structure to systematically generate messages containing invalid data among valid data [44]. It can also use a model describing the behavior of an attacker to drive the test generation process [45]. Some tooling are now available to compute fuzzing strategies such as the fuzz test data generator Fuzzino[3], which determines fuzzed test data by applying security test strategies to message arguments from a given correct communication sequence [46,47].

The second main approach is based on properties or schema languages. Many formalisms have already been used to drive the test generation from a property, or by means of a test purpose. By using this kind of formalisms, the test objectives are expressed either as a particular sequencing of the actions (temporal view) or as properties that the data of the system have to verify (spatial view). Such formalism can address a specific security aspect such as access control domain like OrBac [48] or SPL [49] languages.

Temporal logics, such as the Linear Temporal Logic (LTL) [50,51] allow to specify properties on the state of the system under test w.r.t. several successive moments in its life. Tests can then be obtained using a model-checker in the shape of traces from a model that contradicts the required properties (see [52,53] for example). Input/Output Labelled Transition System (IOLTS) and Input/Output Symbolic Transition System (IOSTS) have been also frequently used to specify test purposes [54,55]. These formalisms enable to specify sequencing of actions by using the actions of the model, and possess two trap states named *Accept* and *Refuse*. The *Accept* states are used as end states for the test generation, while the *Refuse* states allow to cut the traces not targeted by the test generation objectives. For example, these formalisms are used in tools such as TGV [54], STG [56], TorX [57] or Agatha [58]. Another approach, described in [59], consists to generate traces using model-checking techniques from a model specified as an IOLTS, in which a fault have been injected by a mutation operator, according to a fault model. The trace is then used as a test objective for the TGV tool.

[3] `https://github.com/fraunhoferfokus/Fuzzino`

Some security testing approaches are indeed based on the definition of scenarios as test objectives. In [60,61], test cases are issued from UML diagrams as a set of trees. The scenarios are extracted by a breadth-first search on the trees. A similar approach is implemented in the tool *Telling TestStories* [62], which defines a test model from elementary test sequences composed of an initial state, a *test story* and test data. An operational language to describe test schemas in a "textual" way is proposed in [63]. Let us also cite Tobias tool [64,65] that provides a combinatorial unfolding of some given test schemas. The schemas are sequences of patterns composed of operation calls and parameter constraints. The schemas are unfolded independently from any model, therefore the obtained test cases have to be instantiated on a model. In [66], a connection between Tobias and the UCASTING tool is studied to produce instantiated test cases. UCASTING [67] aims to concretize sequences of operations that are derived from a UML model, and thus are not, or only partially, instantiated.

It should also be noted that Advanced Open Standards of the Information Society[4] (OASIS) proposes some works to normalize the description as eXtensible Access Control Markup Language (XACML) or the Security Assertion Markup Language (SML).

4.3 Example of Properties Description Language

To facilitate the use of temporal properties by validation engineers, M. Dwyer et al. have identified in [68] a set of design patterns that allow to express a set of temporal requirements frequently met in industrial studies as temporal properties. A web version of the evolution of this works can be found at `http://patterns.projects.cis.ksu.edu/`. As depicted in Figure 23, a property pattern can be defined by the one way using occurrence patterns. This family is composed of (i) Absence: an event never occurs, (ii) Existence: an event occurs at least once, (iii) Bounded Existence has 3 variants: an event occurs k times, at least k times or at most k times, and (iv) Universality: an event/state is permanent. The second way concerns the order patterns: (v) Precedence: an event P is always preceded by an event Q, (vi) Response: an event P is always followed by an event Q, (vii) Chain Precedence: a sequence of events $P_1, \ldots,$ P_n is always preceded by a sequence $Q_1, \ldots, Q_m$ (it is a generalization of the Precedence pattern), (viii) Chain Response: a sequence of events $P_1, \ldots, P_n$ is always followed by a sequence $Q_1, \ldots, Q_m$ (it defines a generalization of the Response pattern).

From this work, an extension is proposed in [69] to add five scopes. Basically, a scope concerns the pattern observation and is composed of events. Events corresponds to all methods specified in the test model. The interest of the method is that the properties are translated into automata and coverage criteria are proposed to drive the test generation to derive test cases that target the related security patterns.

[4] `http://www.oasis-open.org`

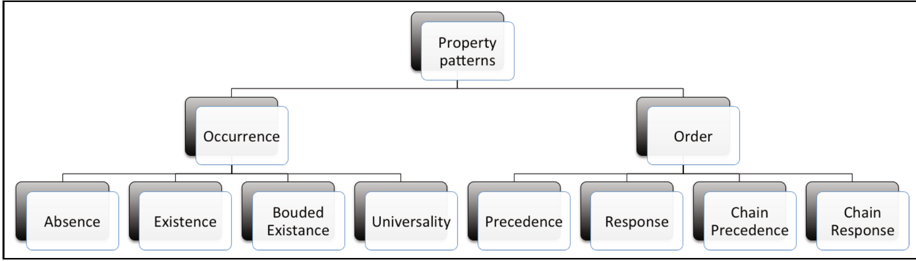


Fig. 23. Pattern expressiveness by M. Dwyer

4.4 Example of Pattern-Driven Security Testing Approach

The purpose of this section is to present an example of Model-Based Testing approach driven by security test pattern [70]. This example aims to validate the detection of SBS-1 malicious signals, formatted according to the ADS-B air-traffic control standard[5], which could be received by the control tower from the aircraft. The ADS-B air-traffic control standard is all about communications between aircraft, and also between aircraft and ground by providing every second a broadcast of the aircraft status (including position, identity, velocity,... calculated using a Global Navigation Satellite System) and make it possible to generate a precise air picture for air traffic management. However, the ADS-B standard is public and all the transmitted information are unencrypted, and decoding them is not difficult (see Figure 24 in which each line defines a separate message that have been sent by a single aircraft).


```
1  "2013/09/16","10:54:09.250","3951363","3C4B03","BER28N","Germany","0","39950","39950","48.21606","6.31798","0","0","466.3","355.7","21300","5334"
2  "2013/09/16","10:54:09.296","4494058","4492EA","BEL8LS","Belgium","0","27000","27000","47.21593","6.40017","0","0","374.3","189.7","273","0111"
3  "2013/09/16","10:54:09.296","172462","02A1AE","LBT5125","Tunisia","0","39000","39000","47.10008","6.34110","64","64","396.4","191.8","274","0112"
4  "2013/09/16","10:54:09.703","3956993","3C6101","HAY2213","Germany","0","38050","38050","47.56545","6.19537","-64","-64","451.1","359.0","21301","5335"
5  "2013/09/16","10:54:09.703","172462","02A1AE","LBT5125","Tunisia","0","39000","39000","47.09935","6.34090","64","64","396.4","191.8","274","0112"
6  "2013/09/16","10:54:09.703","3746549","392AF5","AFR1127","France","0","34400","34400","47.51651","5.61964","-896","-896","407.0","309.5","563","0233"
7  "2013/09/16","10:54:09.906","4738137","484C59","TRA772","Netherlands","0","38000","38000","48.36003","6.21298","-64","-64","464.3","358.0","4423","1147"
8  "2013/09/16","10:54:09.906","3951073","3C49E1","CFG011","Germany","0","36000","36000","47.29768","6.40454","-64","-64","453.1","356.0","21328","5350"
9  "2013/09/16","10:54:09.906","172497","02A1D1","TAR546","Tunisia","0","34350","34350","48.66818","6.18162","-1984","-1984","480.2","358.2","26465","6761"
10 "2013/09/16","10:54:10.187","4921467","4B187B","","Switzerland","0","31000","31000","48.11934","5.62534","64","64","486.7","109.4","0","    "
```


Fig. 24. Excerpt of ADS-B/SBS-1 data stream

In this context, the Model-Based Security Testing generation process aims to produce communication sequences including malicious data. The objective of these generated sequences is to evaluate the vulnerability detection rate of automated air-control system, and the corresponding human attitude during monitoring. It also can be relevant to develop and elaborate new warning protocol, and to improve existing countermeasures, which are today mainly based on data comparison between ADS-B and radar information, and to a latter extent visual inspection. The global process of the approach is depicted in Figure 25.

[5] `http://adsb.tc.faa.gov/ADS-B.htm`

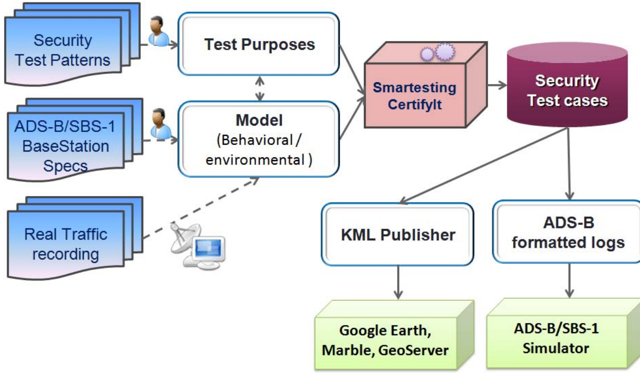


Fig. 25. Security testing process overview

The proposed process is based on the Smartesting Model-Based Testing tool (namely CertifyIt) [27] provided by the company Smartesting[6], which allows to generate test sequences from UML behavioural models and security test purposes, which are described in a language formalizing textual test patterns.

The behavioural model (UML class diagram, object diagram and state diagram with OCL constraints as introduced in the eCinema example in Sect. 3.3) defines the environmental aspects of the domain to be tested in order to generate consistent (from a functional point of view) sequences of ADS-B signals. On the one hand, it includes the communication format of the SBS-1 message of the ADS-B standard (static aspect), and on the other hand, it captures real (or realistic) air-traffic scenarios (dynamic aspect).

A test purpose is here a high-level expression that formalizes a test intention linked to a testing objective to drive the automated test generation on the behavioral model. This is a textual language, which has been originally designed to drive model-based test generation for security components, typically Smart card applications and cryptographic components [71]. This test purpose language has also been extended to be able to formalize typical vulnerability test patterns for Web applications [72]. In the context of this case-study, this test purpose language allows the formalization of attack patterns in terms of states to be reached and SBS-1 messages to be sent. It relies on combining keywords and instructions allowing updating and/or falsifying the real air-traffic scenarios described in the UML behavioural model. The test generation algorithm, computed by the Smartesting CertifyIt tool, enables then to produce mutated real air-traffic scenarios (sequences of transmitted ADS-B signals) by changing and/or adding communication data, which simulate a malicious aircraft broadcast. As example, from a real air-traffic configuration, test patterns and corresponding test purposes can give rise to the production of vulnerability air-traffic scenarios including injection of fake aircrafts into a real configuration, injection of cancelled flights into a real configuration, introduction of (slight) variations in real flights, change of an apparent airliner into fighter(s),...

[6] http://www.smartesting.com

Each of such generated scenarios is typically an abstract sequence of high-level actions from the UML models. These generated test sequences contain the sequence of stimuli, i.e. all the SBS-1 messages sent by the aircrafts concerning their position. These generated sequences, that constitute attack scenarios, are next translated into SBS-1 Simulator using ADS-B formatted signals to be executed on a realistic test bench. They are also concretized into KML language scripts in order to be simulated using simulation tools such as Google Earth, Marble or GeoServer.

Figure 26 shows an example of Google Earth simulation, in which a fake aircraft (red path) has been added to the real air-traffic configuration. Figure 27 shows an excerpt of the falsified ADS-B/SBS-1 data stream, which is automatically generated by the test generator.

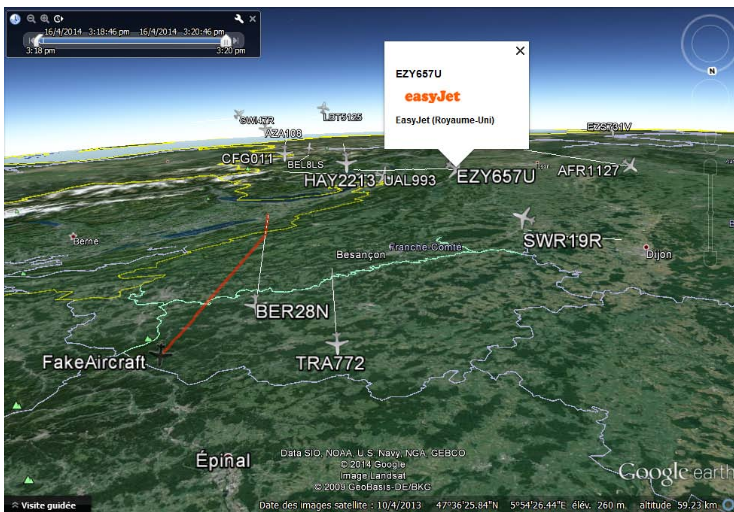


Fig. 26. Simulation of a falsified air-traffic scenario using Google Earth

```
1 "2013/09/16","10:54:09.250","3951363","3C4B03","BER28N","Germany","0","39950","39950","48.21606","6.31798","0"
2 "2013/09/16","10:54:09.250","3951364","3C4B04","FAK123","France","0","34400","34400","47.51651","5.61964","0",
3 "2013/09/16","10:54:09.296","4494058","4492EA","BEL8LS","Belgium","0","27000","27000","47.21593","6.40017","0"
4 "2013/09/16","10:54:09.296","172462","02A1AE","LBT5125","Tunisia","0","39000","39000","47.10008","6.34110","64
5 "2013/09/16","10:54:09.703","3956993","3C6101","HAY2213","Germany","0","38050","38050","47.56545","6.19537","-
```

Fig. 27. Excerpt of a falsified ADS-B/SBS-1 data stream

5 Conclusion

Testing is nowadays a strategic activity at the heart of software quality assurance, no matter the type of software development: all developments undergo some testing, and effort as well as budget are allocated to this task. This paper gave an

overview of existing techniques and state-of-the-art about Model-Based Testing and its deployment to address both functional and security testing.

The idea of Model-Based Testing is to use an explicit abstract model of the system under test and/or of its environment to automatically derive test cases: the behavior of the model is interpreted as the intended behavior of the system under test. The algorithms, driving the test generation process and selecting the test data, enable to ensure a given coverage of the model entities, and so of the functional features of the system. It should be noted that these algorithms mainly originate from structural testing strategies: they are no more applied to the code of the system, but to models specifying its expected functional behavior.

Therefore, Model-Based Testing promises higher quality and conformance to the respective functional safety and quality standards at a reduced cost through increased coverage, advanced test generation techniques, increased automation of the process, eased regression testing management, and finally decreased test maintenance effort. The technology of automated model-based test case generation has matured to the point where large-scale deployments of this technology are becoming commonplace, and a wide range of commercial and open-source tools are now available (this list is not exhaustive !):

- CertifyIt (Smartesting)
- Fokus!MBT (Fraunhofer Fokus)
- MaTeLo (All4Tec)
- ModelJUnit (CSZ)
- Conformiq Designer (Conformiq)
- Reactis (Reactive System)
- Scade (Esterel Technologies)
- Spec Explorer (Microsoft)
- STG - TGV (IRISA)
- UPPAAL Cover (UP4ALL)

Even if Model-Based testing approach is an effective and useful technique, which brings significant progress in the current practice of functional software testing, it does not solve all testing problems. This weakness especially occurs when addressing non functional testing such as security testing that aims at validating and verifying that a software system meets its security requirements. Indeed, contrary to behavioral features, test objectives targeting security requirements cannot be easily derived from the structure of the test model, and the expertise of the security engineers is clearly missing. To tackle this weakness, especially regarding security issues, dedicated test model targeting security aspects (including risk assessment results) and specific testing strategies have been created. These strategies are mainly based on fuzzing algorithms and security test patterns languages. These artefacts drive the security test generation process, and therefore replace the coverage criteria traditionally used to address functional purposes. Although model-based approaches for security testing are not yet so advanced and so popular compared to functional Model-Based Testing approaches, this research direction gives rise, from several years, to efficient and emerging approaches and technologies, especially concerning fuzzing techniques.

References

1. IEEE: IEEE Standard for Software and System Test Documentation. IEEE Std 829-2008 (2008)
2. Myers, G., Sandler, C., Badgett, T., Thomas, T.: The Art of Software Testing, 2nd edn. Wiley (2004) ISBN: 978-0-4714-6912-4
3. Dijkstra, E.: Notes on structured programming. Technical Report EWD249, Eindhoven University of Technology (1970)
4. Tretmans, J.: Model Based Testing with Labelled Transition Systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
5. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. IEEE Transactions on Software Engineering 15(11), 1318–1332 (1989)
6. Zhu, H., Hall, P., May, J.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29(4), 366–427 (1997)
7. Vilkomir, S., Bowen, J.: Formalization of software testing criteria using the Z notation. In: Proceedings of the 25th International Conference on Computer Software and Applications (COMPSAC 2001), Chicago, USA. IEEE Computer Society Press (October 2001)
8. Offutt, A., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 1999), pp. 119–131. IEEE Computer Society Press, Las Vegas (1999)
9. Chilenski, J., Miller, S.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 9(5), 193–200 (1994)
10. RTCA Committee SC-167: Software considerations in airborne systems and equipment certification, 7th draft to Do-178B/ED-12A (July 1992)
11. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems, 2nd edn. John Wiley & Sons, New York (1995)
12. Legeard, B., Kosmatov, N., Peureux, F., Utting, M.: Boundary Coverage Criteria for Test Generation from Formal Models. In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004), Saint-Malo, France, pp. 139–150. IEEE Computer Society Press (November 2004)
13. Prowell, S.J.: Jumbl: A tool for model-based statistical testing. In: HICSS, p. 337 (2003)
14. Bauer, T., Bohr, F., Landmann, D., Beletski, T., Eschbach, R., Poore, J.: From requirements to statistical testing of embedded systems. In: SEAS 2007: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems, p. 3. IEEE Computer Society, Washington, DC (2007)
15. le Guen, H., Marie, R.A., Thelin, T.: Reliability estimation for statistical usage testing using markov chains. In: ISSRE, pp. 54–65. Computer Society (2004)
16. Offutt, A., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. The Journal of Software Testing, Verification and Reliability 13(1), 25–53 (2003)
17. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11-11 standard case study. Software: Practice and Experience 34(10), 915–948 (2004)
18. Zhu, H., Belli, F.: Advancing test automation technology to meet the challenges of model-based software testing. Journal of Information and Software Technology 51(11), 1485–1486 (2009)

19. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Elsevier Science (2006) ISBN 0 12 372501 1
20. Dias-Neto, A., Travassos, G.: A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges. Advances in Computers 80, 45–120 (2010) ISSN: 0065-2458
21. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability 22(5), 297–312 (2012)
22. OMG: Sysml documentation, http://www.omgsysml.org/
23. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire (1992)
24. Abrial, J.R.: The B-Book. Cambridge University Press (1996)
25. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
26. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A subset of precise UML for model-based testing. In: A-MOST 2007, 3rd Int. Workshop on Advances in Model Based Testing, pp. 95–104. ACM Press (2007)
27. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F.: A test generation solution to automate software testing. In: 3rd Int. Workshop on Automation of Software Test, AST 2008, Leipzig, Germany, pp. 45–48. ACM Press (May 2008)
28. Schieferdecker, I., Großmann, J., Schneider, M.: Model-Based Security Testing. In: Proceedings of the 7th Int. Workshop on Model-Based Testing (MBT 2012), Tallinn, Estonia. EPTCS, vol. 80, pp. 1–12 (March 2012)
29. Tian-yang, G., Yin-sheng, S., You-yuan, F.: Research on Software Security Testing. World Academy of Science, Engineering and Technology 4(9), 572–576 (2010)
30. Wichers, D.: Owasp top 10 (October 2013), https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (last visited: May 2014)
31. MITRE: Common weakness enumeration (October 2013), http://cwe.mitre.org/ (last visited: May 2014)
32. Whitehat: Website security statistics report (October 2013), https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf (last visited: May 2014)
33. Felderer, M., Agreiter, B., Zech, P., Breu, R.: A classification for model-based security testing. In: The Third International Conference on Advances in System Testing and Validation Lifecycle, VALID 2011, pp. 109–114 (2011)
34. Lund, M.S., Solhaug, B., Stølen, K.: Model-Driven Risk Analysis: The CORAS Approach, 1st edn. Springer Publishing Company, Incorporated (2010)
35. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
36. Jürjens, J.: Model-based security testing using UMLsec. Electron. Notes Theor. Comput. Sci. 220(1), 93–104 (2008)
37. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based modeling language for model-driven security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
38. Bosik, B.S., Uyar, M.U.: Finite state machine based formal methods in protocol conformance testing: from theory to implementation. Computer Networks and ISDN Systems 22(1), 7–33 (1991); 9th IFIP TC-6 International Symposium on Protocol Specification, Testing and Verification

39. Fernandez, J.-C., Jard, C., Jeron, T., Viho, C.: An experiment in automatic generation of test suites for protocols with verification technology. Science of Computer Programming 29(1), 123–146 (1997)
40. Dadeau, F., Héam, P.C., Kheddam, R.: Mutation-based test generation from security protocols in HLPSL. In: Harman, M., Korel, B. (eds.) 4th Int. Conf. on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, pp. 240–248. IEEE Computer Society Press (March 2011)
41. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
42. Godefroid, P.: Random testing for security: blackbox vs. whitebox fuzzing. In: Proceedings of the 2nd International Workshop on Random Testing: Co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 1. ACM (2007)
43. Miller, B.P., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. Commun. ACM 33(12), 32–44 (1990)
44. Takanen, A., DeMott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, Inc., Norwood (2008)
45. Duchene, F., Groz, R., Rawat, S., Richier, J.L.: XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing. In: Proc. of the 5th Int. Conference on Software Testing, Verification and Validation (ICST 2012), Montreal, Canada, pp. 815–817. IEEE CS (April 2012)
46. Schieferdecker, I.: Model-Based Fuzzing for Security Testing. Keynote talk at the 3rd International Workshop on Security Testing (SECTEST 2012), Montreal, Canada (April 2012)
47. Schneider, M., Großmann, J., Tcholtchev, N., Schieferdecker, I., Pietschker, A.: Behavioral Fuzzing Operators for UML Sequence Diagrams. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) SAM 2012. LNCS, vol. 7744, pp. 88–104. Springer, Heidelberg (2013)
48. Abou El Kalam, A., El Baida, R., Balbiani, P., Benferhat, S., et al.: Organization based access control. In: Lutfiyya, H., Moffett, J., Garcia, F. (eds.) Policies for Distributed Systems and Networks (POLICY 2003), Como, January 01-December 31, pp. 120–131. Institute of Electrical and Electronics Engineers (2003)
49. Ribeiro, C., Zuquete, A., Ferreira, P., Guedes, P.: Spl: An access control language for security policies and complex constraints. In: NDSS, vol. 1 (2001)
50. Pnueli, A.: The temporal semantics of concurrent programs. Theoretical Computer Science 13, 45–60 (1981)
51. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In: IEEE Int. Conf. on Information Reuse and Integration, IRI 2004, pp. 413–498 (November 2004)
52. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. SIGSOFT Softw. Eng. Notes 24(6), 146–162 (1999)
53. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: 2nd IEEE Int. Conf. on Formal Engineering Methods, ICFEM 1998, pp. 46–54. IEEE Computer Society Press (December 1998)
54. Jard, C., Jéron, T.: 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)
55. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)

56. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: A symbolic test generation tool. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 151–173. Springer, Heidelberg (2002)
57. Tretmans, G.J., Brinksma, H.: TorX: Automated model-based testing. In: First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp. 31–43 (December 2003)
58. Bigot, C., Faivre, A., Gallois, J.-P., Lapitre, A., Lugato, D., Pierron, J.-Y., Rapin, N.: Automatic test generation with AGATHA. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 591–596. Springer, Heidelberg (2003)
59. Aichernig, B.K., Weiglhofer, M., Wotawa, F.: Improving fault-based conformance testing. Electron. Notes Theor. Comput. Sci. 220, 63–77 (2008)
60. Bertolino, A., Marchetti, E., Muccini, H.: Introducing a reasonably complete and coherent approach for model-based testing. Electron. Notes Theor. Comput. Sci. 116, 85–97 (2005)
61. Basanieri, F., Bertolino, A., Marchetti, E.: The Cow_Suite approach to planning and deriving test suites in UML projects. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 383–397. Springer, Heidelberg (2002)
62. Felderer, M., Breu, R., Chimiak-Opoka, J., Breu, M., Schupp, F.: Concepts for Model-based Requirements Testing of Service Oriented Systems. In: Proceedings of the IASTED International Conference, vol. 642, p. 018 (2009)
63. Fourneret, E., Ochoa, M., Bouquet, F., Botella, J., Jurjens, J., Yousefi, P.: Model-based security verification and testing for smart-cards. In: 6th International Conference on Availability, Reliability and Security, ARES 2011, pp. 272–279. IEEE (2011)
64. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS combinatorial test suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 281–294. Springer, Heidelberg (2004)
65. Ledru, Y., Dadeau, F., Du Bousquet, L., Ville, S., Rose, E.: Mastering combinatorial explosion with the TOBIAS-2 test generator. In: ASE 2007: Procs of the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering, pp. 535–536 (2007)
66. Maury, O., Ledru, Y., du Bousquet, L.: Intégration de TOBIAS et UCASTING pour la génération des tests. In: 16th Int. Conf. on Software and Systems Engineering and their Applications, ICSSEA 2003, Paris, France (2003)
67. Van Aertryck, L., Jensen, T.: UML-CASTING: Test synthesis from UML models using constraint resolution. In: AFADL 2003 (2003)
68. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21st International Conference on Software Engineering, ICSE 1999, Los Angeles, California, United States, pp. 411–420 (1999)
69. Castillos, K.C., Dadeau, F., Julliand, J., Kanso, B., Taha, S.: A compositional automata-based semantics for property patterns. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 316–330. Springer, Heidelberg (2013)
70. Botella, J., Cao, P., Civeit, C., Gidoin, D., Peureux, F.: Model-Based Test Generation of Aircraft Traffic Attack Scenarios using ADS-B Standard Signals. In: 1-st User Conference on Advanced Automated Testing, UCAAT 2013, Paris, France (October 2013)
71. Botella, J., Bouquet, F., Capuron, J.F., Lebeau, F., Legeard, B., Schadle, F.: Model-Based Testing of Cryptographic Components – Lessons Learned from Experience. In: Proc. of the 6th Int. Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg, pp. 192–201. IEEE CS (March 2013)
72. Lebeau, F., Legeard, B., Peureux, F., Vernotte, A.: Model-Based Vulnerability Testing for Web Applications. In: Proc. of the 4th Int. Workshop on Security Testing (SECTEST 2013), Luxembourg, pp. 445–452. IEEE CS Press (March 2013)