SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2010; **00**:1–15
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

A taxonomy of model-based testing approaches

Mark Utting[1]*, Alexander Pretschner[2] and Bruno Legeard[3]

[1]*School of Computing and Mathematical Sciences, University of Waikato, New Zealand*
[2]*Certifiable Trustworthy IT Systems, Karlsruhe Institute of Technology, Germany*
[3]*Smartesting and Laboratoire d'Informatique de l'Université de Franche-Comté, Besançon, France*

SUMMARY

Model-based testing relies on models of a system under test and/or its environment to derive test cases for the system. This article discusses the process of model-based testing and defines a taxonomy that covers the key aspects of model-based testing approaches. It is intended to help with understanding the characteristics, similarities and differences of those approaches, and with classifying the approach used in a particular model-based testing tool. To illustrate the taxonomy, a description of how three different examples of model-based testing tools fit into the taxonomy is provided. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: model-based testing approaches, taxonomy, survey

1. INTRODUCTION

Testing aims at showing that the intended and actual behaviours of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: finding observable differences between the behaviour of the implementation and the intended behaviour of the system under test (SUT), as expressed by its requirements.

Model-based testing (MBT) is a variant of testing that relies on explicit behaviour models that encode the intended behaviours of a SUT and/or the behaviour of its environment. Test cases are generated from one of these models or their combination, and then executed on the SUT. The use of explicit models is motivated by the observation that traditionally, the process of deriving tests tends to be unstructured, not reproducible, not documented, lacking detailed rationales for the test design, and dependant on the ingenuity of single engineers. The idea is that artifacts that explicitly encode the intended SUT and possibly environment behaviours can help mitigate these problems.

The ideas of model-based testing, then dubbed specification-based testing, date back to the Seventies [1]. Recent emphasis on model-based and test-centered development methodologies as well as the level of maturity of technology from the area of formal verification have led to a strong increased interest in the subject in the past decade, both in the academic field and in the industry. Recent surveys by Hierons et al. [2] as well as Dias-Neto et al. [3] provide a comprehensive overview of the abundant technical literature in the MBT field. Dias-Neto et al. analyse 271 papers and count more than 219 different MBT approaches that have been proposed, often with associated tools. With so many approaches in the field, it can be a daunting task for a practitioner or researcher to make sense of the myriads of technical proposals. This can stifle the adoption of model based testing technology in industry and limit the further improvement of MBT approaches.

*Correspondence to: School of Computing and Mathematical Sciences, University of Waikato, Private Bag 3105, Hamilton, New Zealand. E-Mail: marku@cs.waikato.ac.nz

Contribution. This paper helps by developing a taxonomy of six essential dimensions that characterize the different model based testing approaches, with examples of how several typical tools fit into that taxonomy. A classification of many more commercial and academic MBT tools using this taxonomy is available from a website [4]. The focus of this article is on MBT for functional testing, given that this is currently the main industrial usage of MBT. Moreover, the taxonomy is oriented towards users of model-based testing. It provides a framework for comparing and qualitatively assessing tools and techniques.

Intended limitations. The perspective on MBT adopted in this paper is inherently bound to the notion of *choice*: tools generate tests from test models. These test models, by their very nature, do not specify single tests but rather sets of possible tests, and it is up to a MBT tool to choose tests from this set (Section 3.5). This is why keyword-based testing or test description languages and tools [5] are not included in this article, even though those approaches are sometimes called model-based as well.

This paper provides concepts and a frame of reference for assessing model-based tools. Evaluating the pragmatics of MBT tools, such as their ease of use, speed, interoperability, support for evolving requirements (e.g., generating tests for the subset of the requirements that have changed), or support for traceability (i.e., relating the generated tests back to the model, or even back to the informal system requirements) is important in practice. However, these issues are shared by many kinds of software engineering tools and are independent of the dimensions in this taxonomy.

Research results in more theoretical aspects of model-based testing are outside the scope of this article. Moreover, while the article contains an overview of the process of model-based testing (see Section 2), it deliberately does not discuss the empirical evidence for the effectiveness of model-based testing or its limitations, since this has been done elsewhere [6, Chapter 2]; see also references [7, 8, 9, 10, 11, 12, 13, 14]. Detailed examples of modeling and test generation using several kinds of model-based testing tools are provided in the literature [6, 15].

Organisation. Section 2 introduces the fundamental concepts of model-based testing along with the terminology used. Section 3 describes the taxonomy, which is used in Section 4 to classify a collection of model-based testing tools in an exemplary manner. Section 5 discusses related work, and Section 6 draws conclusions.

2. PROCESS AND TERMINOLOGY

This section is used to fix terminology and to describe the general process of model-based testing.

A *test suite* is a finite set of test cases. A *test case* is a finite structure of input and expected output: a pair of input and output in the case of deterministic transformative systems, a sequence of input and output in the case of deterministic reactive systems, and a tree or a graph in the case of non-deterministic reactive systems. The input part of a test case is called *test input*.

Model-based testing encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases. The assumption here is that models are specified with languages that are sufficiently precise to allow, in principle, a machine to derive tests from these models.

A generic process of model-based testing then proceeds as follows (Figure 1).

Step 1. A model of the SUT is built from informal requirements or existing specification documents. This model is often called a *test model*, because the abstraction level and the focus of the model is directly linked with the testing objectives. In some cases, the test model could also be the design model of the system under test, but it is important to have some independence between the model used for test generation and any development models, so that errors in the development model are not propagated into the generated tests [16]. For this reason, it is usual either to develop a test-specific model directly from the informal requirements, or to reuse just a few aspects of the development model as the basis for a test model, which is then validated against the informal requirements. Validating the model means that the requirements themselves are scrutinised for consistency and completeness, and this often exposes requirements errors.

DOI: 10.1002/stvr

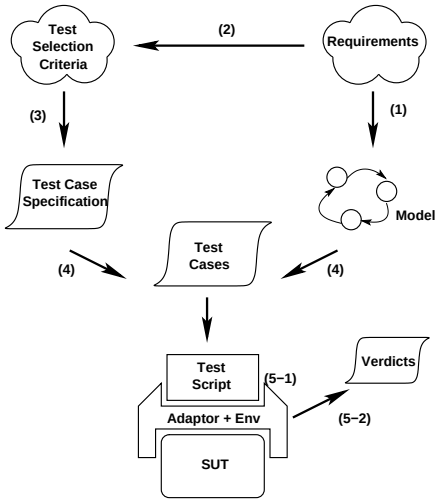

Figure 1. The Process of Model-Based Testing

In terms of model-based testing, the necessity of validating the model implies that the model must be simpler (more abstract) than the SUT, or at least easier to check, modify and maintain. Otherwise, the efforts of validating the model would equal the efforts of validating the SUT. Throughout this paper, the term 'abstraction' will be used to denote both the deliberate omission of detail in the model and the encapsulation of details by means of high-level language constructs (see Section 3.1). The test model can reside at various levels of abstraction. The most abstract variant maps each possible input to the output 'no exception' or 'no crash'. It can also be abstract in that it neglects certain functionality, or disregards certain quality-of-service attributes such as timing or security (Section 3.1; [17]).

On the other hand, the model must be sufficiently precise to serve as a basis for the generation of 'meaningful' test cases. This means that the tests generated from the model should be complete enough in terms of actions, input parameters and expected results to provide real added value. If not, the test design job still has to be done manually, and there is little added value in generating tests from the model.

Step 2. Test selection criteria are chosen, to guide the automatic test generation so that it produces a 'good' test suite – one that fulfills the test policy defined for the SUT. Defining a clear test policy and test objectives for a system and associated development project is part of all testing methods such as TMap® [18] or the ISTQB guidelines [19] that are widely used in industry. In such methods, the test policy and test objectives are formalized into Test Plan documents, which define the scope of testing and the various testing strategies and techniques that will be used in the project for each testing level (e.g. unit testing, integration testing, system testing, acceptance testing).

Test selection criteria can relate to a given functionality of the system (requirements-based test selection criteria), to the structure of the test model (state coverage, transition coverage, def-use dataflow coverage), to data coverage heuristics (pair-wise, boundary value), to stochastic characterisations such as pure randomness or user profiles, to properties of the environment, and they can also relate to a well-defined set of faults.

Step 3. Test selection criteria are then transformed into *test case specifications*. Test case specifications formalise the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite (see Step 4). For instance, 'state coverage' of a finite state machine (FSM) might translate into a set of test case specifications such as $\{reach\ s0, reach\ s1, reach\ s2, \ldots\}$, where $s0, s1, s2, \ldots$ are all the states of the FSM. A test case specification is a high level description of a desired test case.

DOI: 10.1002/stvr

Step 4. Once the model and the test case specifications are defined, a set of test cases is *generated*, with the aim of satisfying all the test case specifications. The set of test cases that satisfy a test case specification with respect to the model can be empty, in which case the test case specification is said to be *unsatisfiable*. Usually, however, there are several or many test cases that satisfy it, and the test case generator will choose just one of those test cases. Some test generators may spend significant effort in minimizing the test suite, so that a small number of generated test cases cover a large number of test case specifications.

Step 5. Once the test suite has been generated, the test cases are *run*. Test execution may be manual - i.e. by a physical person - or may be automated by a *test execution environment* that provides facilities to automatically execute the tests and record test verdicts. Sometimes, especially for non-deterministic systems, the generation and running of the tests are dove-tailed together, which will be called *online* testing in this article.

Running a test case includes several steps. Recall that model and SUT reside at different levels of abstraction, and that these different levels must be bridged [16]. For example, an abstract test case for a bookshop website might be $checkPrice(WarAndPeace) = \$19.50$, where $checkPrice$ is the name of the webservice to be used, $WarAndPeace$ is the book to be queried, and \$19.50 is the expected result. *Executing a test case* then starts by concretising the test inputs (e.g., to obtain a detailed web services call) and sending that concrete data to the SUT (see step 5-1 in Fig. 1). Secondly, the resulting concrete output of the SUT (e.g., a page of XML) must be captured and must then be abstracted to obtain the high-level expected result (a price) that can then be compared against the expected result (step 5-2 in Fig. 1). The component that performs the concretisation of test inputs and abstraction of test outputs is called the *adaptor*, because it adapts the abstract test data to the concrete SUT interface.

A *test script* is some executable code that executes a test case, abstracts the output of the SUT, and then builds the verdict. Note that an adaptor is a concept and not necessarily a separate software component—it may be integrated within the test scripts.

To summarize, model-based testing involves the following major activities: building the model, defining test selection criteria and transforming them into operational test case specifications, generating tests, conceiving and setting up the adaptor component (if the generated tests are to be executed automatically, the adaptor is usually a significant proportion of the workload) and executing the tests on the SUT. The model of the SUT is used as the basis for test generation, but also serves to validate requirements and check their consistency.

3. THE TAXONOMY

The definition of the process gives rise to six dimensions of model-based testing approaches. Along with possible instantiations of each dimension, these are presented in this section. The dimensions are largely independent of each other, but not entirely: for instance, if a project is concerned with a continuous rather than a discrete system, this is likely to limit its choice of modelling paradigm, of test selection criteria, and of test case generation technology.

Figure 2 gives an overview of the taxonomy. The 'A/B' alternatives at the leaves indicate mutually exclusive alternatives, while the curved lines indicate alternatives that are not necessarily mutually exclusive (for example, some tools may use more than one test generation technology, and it is common and desirable to support several kinds of test selection criteria).

The choice of these six dimensions directly reflects the process introduced in Section 2. Step 1 (building the model) is reflected by the three dimensions within the *model specification* category: *scope*, *characteristics*, and *modelling paradigm*. Steps 2 and 3 (choosing test selection criteria and building test case specifications) are reflected by the *test selection criteria* dimension within the *test generation* category. Step 4 (generating tests) is reflected by the *Technology* dimension with the *test generation* category. Step 5 (running tests) is reflected by the *on/offline* dimension of the *test execution* category.

Other perspectives that give rise to a taxonomy of model-based testing and that do not start from the process can, of course, also be justified. For instance, one could also start from the different

DOI: 10.1002/stvr

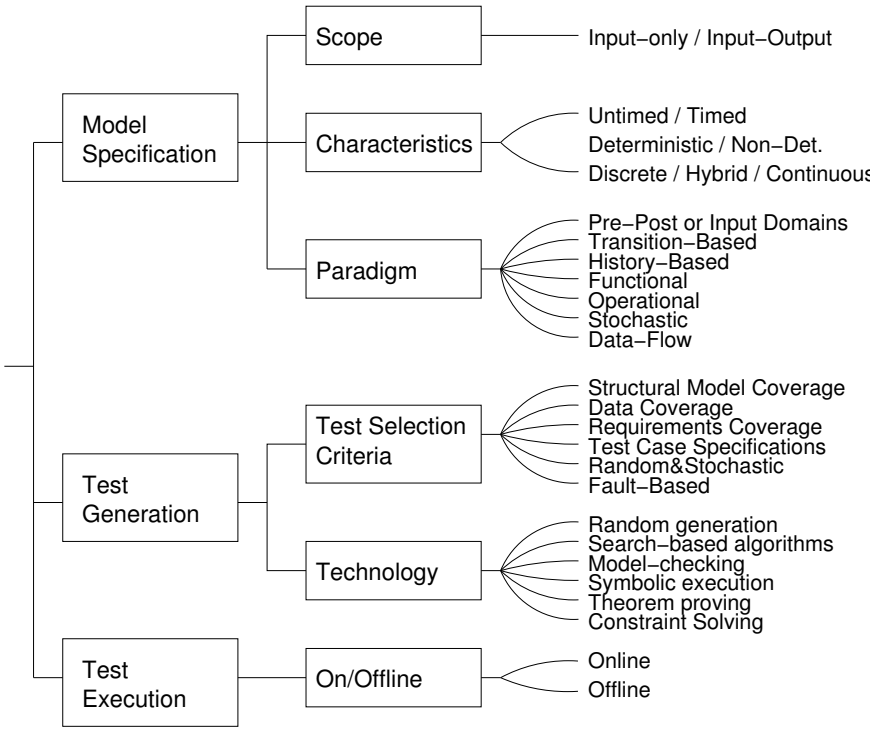

Figure 2. Overview of the Taxonomy

artifacts that are developed or used in that process, e.g., models, test specifications, test drivers, properties, tests, etc. The rationale for the decision to use the process as a basis is that it is easier to agree on the activities of the process, and thus to justify the completeness of the taxonomy, than to agree on the different relevant artifacts. This, of course, does not mean that such a different taxonomy wouldn't be valuable as well.

3.1. Model Scope

The first dimension is the *scope* of the model, which is classified into a binary decision: does the model specify only the *inputs* to the SUT, or does it specify the expected *input-output* behaviour of the SUT? The input-only models are generally easier to specify, but they have the disadvantage that the generated tests will not be able to act as an oracle. The generated tests may implement an implicit 'robustness' oracle, such as checking that the SUT does not crash or throw any exceptions, but they cannot check the correctness of the actual SUT output values, since the model does not specify the expected output values. So input-only models produce weak oracles that are incapable of verifying the correctness of the SUT functional behaviour.

Input-output models of the SUT not only model the allowable inputs that can be sent to the SUT, but must also capture some of the intended behaviour of the SUT. That is, the model must be able to predict in advance the expected outputs of the SUT for each input, or at least be able to check whether an output produced by the SUT is allowed by the model or not.

Input models can be seen as models of the environment. Attacker models in security testing are a prominent example; Markov chains used for statistical testing are a second one. The most abstract model of any environment is one that fully nondeterministically emits all possible inputs to the SUT. An attacker model encodes some possible behaviours of the environment, so is less abstract; specific functionalities can also be encoded in this way [20]. The most concrete, or least abstract, version of an environment model would specify precisely those inputs that can be sent to the SUT, but such very concrete environment models rarely occur in practice.

DOI: 10.1002/stvr

Similarly, the model of the SUT can be provided at different levels of abstraction. The most abstract version is found in robustness testing and need not be specified explicitly: the model that reacts to any input with 'no exception thrown.' More concrete versions specify some of the behavior to be tested [17], and models that are as precise as the implementation are also conceivable, though rare because of the cost of developing and validating them.

In practice, whenever a MBT model has input-output scope, it specifies some aspects of the environment and also some aspects of the SUT, possibly at differing levels of abstraction.

3.2. Model Characteristics

Model characteristics relate to the incorporation of timing issues, to nondeterminism, and to the continuous or event-discrete nature of the model. These model characteristics are typically chosen based on what kind of SUT is being tested.

Timing issues are particularly relevant in the large class of real-time systems. Because of the additional degree of freedom, these systems are notoriously hard to test. Applying the ideas of model-based testing to real-time systems is the subject of intense research activities [21].

Nondeterminism can occur in the model and/or the SUT. If the SUT exhibits jitter in the time or value domains, this can often be handled when the verdict is built (which might be possible only after all input was applied). If the SUT exhibits genuine nondeterminism, as a consequence of concurrency, for instance, then it is possible that test stimuli as provided by the model depend on prior reactions of the SUT. In these cases, the non-determinism must be catered for by the model, and also by the test cases (they are not sequences anymore, but rather trees or graphs). Finally, nondeterminism in the model can be used for testing deterministic systems. One example is using non-deterministic timeouts to avoid a detailed timing model (e.g., [8, p. 395]).

In terms of dynamics, models can be discrete, continuous or a mixture of the two (hybrid). Most work in model-based testing has focused on event-discrete systems, but continuous or hybrid models are often common in many embedded systems. Like model-based real-time testing, testing continuous systems is the subject of on-going research [21].

The distinction between different characteristics is important, because it impacts the choice of the modelling paradigm, technology for test case generation, and the interleaving of generating and executing tests (online versus offline).

3.3. Model Paradigm

The third dimension is what paradigm and notation are used to describe the model. There are many different modelling notations that have been used for modelling the behaviour of systems for test generation purposes. It is convenient to group them into the following paradigms, adapted from van Lamsweerde [22]. These paradigms are also used in the MBT state of the art survey provided by Dias-Neto et al. [3]. The overview article of Hierons et al. [2] discusses many of these paradigms in more detail.

State-Based (or Pre/Post) Notations. These model a system as a collection of variables, which represent a snapshot of the internal state of the system, plus some operations that modify those variables. Each operation is usually defined by a *precondition* and a *postcondition*, or the postcondition may be written as explicit code that updates the state. Examples of these notations include Z, B, VDM, JML, OCL, and the C#-plus-preconditions used by Spec Explorer (Section 4).

In the special case where a pre/post notation is used in an input-only model, there is obviously no postcondition, so the pre/post notation is reduced to describing the domains of the input variables and the relationships between variables. This is called an *Input-Domain* notation–it is widely used by pairwise testing tools, for example AETG [23].

Transition-based Notations. These focus on describing the *transitions* between different states of the system. Typically, they are graphical node-and-arc notations, like finite state machines (FSMs), where the nodes of the FSM represent the major states of the system and the arcs represent the actions or operations of the system. Textual or tabular notations are also used to specify the transitions. In practice, transition-based notations are often made more expressive by adding data variables, hierarchies of machines and parallelism between machines. Examples of transition-based

DOI: 10.1002/stvr

notations used for MBT include FSMs themselves, statecharts (e.g. UML State Machines, Statemate statecharts and Simulink Stateflow charts), labelled transition systems and I/O automata.

History-based Notations. These notations model a system by describing the allowable traces of its behaviour over time. Various notions of time can be used (discrete or continuous, linear or branching, points or intervals etc.), leading to many different kinds of temporal logics.

Message-sequence charts and related formalisms are also included in this group. These are graphical and textual notations for specifying sequences of interactions between components.

Functional Notations. These describe a system as a collection of mathematical functions. The functions may be first-order only, as in the case of algebraic specifications, or higher-order, as in notations like HOL. For an example of the use of algebraic specifications for model-based testing, see the work of Gaudel and LeGall [24].

Operational Notations. These describe a system as a collection of executable processes, executing in parallel. They are particularly suited to describing distributed systems and communications protocols. Examples include process algebras such as CSP or CCS as well as Petri net notations. Slightly stretching this category, hardware description languages like VHDL or Verilog are also included in this category.

Stochastic Notations. These describe a system by a probabilistic model of the events and input values and tend to be used to model environments rather than SUTs. For example, Markov chains are used to model expected usage profiles, so that the generated tests exercise that usage profile.

Data-Flow Notations. These notations concentrate on the data rather than the control flow. Prominent examples are Lustre, and the block diagrams of Matlab Simulink, which are often used to model continuous systems.

In practice, several paradigms can be represented in one single notation. For example, the UML notation offers both a transition-based paradigm, with state machine diagrams, and a pre-post paradigm, with the OCL language. The two paradigms can be used at the same time in a test model. For example, this helps to express both the dynamic behaviour and some business rules on discrete data types. Another example is Matlab, which models embedded real-time systems using a combination of Simulink block diagrams (a data-flow notation) and Stateflow statecharts (a transition-based notation).

3.4. Test Selection Criteria

The fourth dimension defines the facilities that are used to control the generation of tests. MBT tools can be classified according to which kinds of test selection criteria they support. Note that these selection criteria indirectly define properties of the generated test suites, including their fault detection power, cardinality, complexity, etc. While most of these criteria are not unique to model-based testing but also apply to more traditional forms of testing, they are a major distinguishing feature between currently available model-based testing tools, which is why they make for an important dimension of this article's taxonomy. The following subsections briefly review the most commonly-used criteria.

Defining the 'best' criterion is not possible in general. It is the task of the test engineer to configure the test generation facilities and choose adequate test selection criteria and test case specifications to meet the project test objectives, e.g., functionality, robustness, security, or performance.

Structural Model Coverage Criteria. These criteria exploit the structure of the model, such as the nodes and arcs of a transition-based model, or conditional statements in a model in pre/post notation.

The modelling notation often suggests specific kinds of structural coverage criteria. For example, when the model uses a pre-post notation, some coverage criteria that are commonly used are: cause-effect coverage, and coverage of all disjuncts in the postcondition. For algebraic modelling notations, coverage of the axioms is an obvious coverage criterion.

For transition-based models, which use explicit graphs containing nodes and arcs, there are many graph coverage criteria that can be used to control test generation. Some of the coverage criteria commonly used are all nodes (that is, all states), all transitions, all transition-pairs, and all cycles.

DOI: 10.1002/stvr

The FSM isomorphism-checking methods developed for testing protocols (W-method, Wp-method, D-method etc.) [25, 26] are also based on structural coverage of FSM models.

Another set of structural coverage criteria are useful for exercising complex boolean decisions within models. This same need arises in white box testing (code-based testing), so many of the well-known code-based structural coverage criteria [27, 28] that require certain combinations of atomic conditions and decisions to take certain values, have been adapted to work on models. Similarly, many data-flow coverage criteria [29] for code have been adapted to models. These criteria can be applied to any modelling notation that contains variables (see [6, Chapter 4] for a detailed presentation of structural coverage criteria).

Data Coverage Criteria. These criteria deal with how to choose a few test values from a large data space. The basic idea is to split the data space into equivalence classes and choose one representative from each equivalence class, with the hope that the elements of this class are 'equivalent' in terms of their ability to detect failures. Pairwise and N-way coverage criteria [30] are popular forms of data coverage criteria. For ordered data types, the partitioning of a range of values into equivalence classes is usually complemented by picking extra tests from the boundaries of the intervals. Boundary analysis [31] and domain analysis [32, Chapter 7] are widely accepted as fault detection heuristics and can be used as coverage criteria for test generation (for comparison with random testing, see the respective seminal papers [33, 34, 35, 36] and the summary by Gaston and Seifert [37]).

Requirements-Based Coverage Criteria. When elements of the model can be explicitly associated with informal requirements of the SUT, coverage can also apply to requirements. For example, if requirement numbers are attached to transitions of a UML state machine or to predicates within the postconditions of a pre-post model, then test generation can aim to cover all requirements.

Ad-hoc Test Case Specifications. Explicit test case specifications can obviously be used to control test generation. In addition to the model, the test engineer writes a test case specification in some formal notation, and these are used to determine which tests will be generated. For example, they may be used to restrict the paths through the model that will be tested, to focus the testing on heavily used cases, or to ensure that particular paths will be tested. The notation used to express these test objectives may be the same as the notation used for the model, or it may be a different notation. Notations commonly used for test objectives include UML Sequence diagrams, FSMs, regular expressions, temporal logic formulae, constraints and Markov chains (for expressing intended usage patterns). This family of coverage criteria relates to the scenario-based testing approach (e.g., [38, 39, 40]) where test cases are generated from descriptions of abstract scenarios.

Random and Stochastic Criteria. These are mostly applicable to environment models, because it is the environment that determines the usage patterns of the SUT. The probabilities of actions are modelled directly or indirectly [41, 42]. The generated tests then follow an expected usage profile.

Fault-based Criteria. These are mostly applicable to SUT models, because the goal of testing is to find faults in the SUT. One of the most common fault-based criteria is mutation coverage. This involves mutating the model, then generating tests that would distinguish between the mutated model and the original model. The assumption is that there is a correlation between faults in the model and in the SUT, and between mutations and real-world faults [9, 43].

3.5. Test Generation Technology

One of the most appealing characteristics of model-based testing is its potential for automation. Given the test model and some test case specifications, test cases can be derived stochastically, or by using dedicated graph search algorithms and search-based techniques, model checking, symbolic execution, deductive theorem proving, or constraint solving.

Random generation of tests is done by sampling the input space of a system. In the case of reactive systems, finite traces can be selected randomly by sampling the input space and applying it to the model of the SUT in order to infer the expected output part. A random walk on the model may result in test suites with different characteristics. Random walks can also be performed on

DOI: 10.1002/stvr

environment models given in the form of (stochastic) usage models, and obviously, this results in certain transition probabilities for the SUT [44].

Search-based algorithms for model-based test generation include graph search algorithms such as node or arc coverage algorithms (e.g., the Chinese Postman algorithm [45], which covers each arc at least once), as well as other search-based algorithms such as metaheuristic search, evolutionary algorithms (e.g., genetic algorithms) and simulated annealing. This field has been of burgeoning interest for many researchers in recent years, particularly for automated test data selection [46].

(Bounded) model checking is a technology for verifying or falsifying properties of a system. For certain classes of properties, model checkers can yield counter examples when a property is not satisfied. The general idea of test case generation with model checkers is to first formulate test case specifications as reachability properties, for instance, *'eventually, a certain state is reached, or a certain transition fires'* (e.g., [47, 48]). A model checker then, by searching for counter-examples for the negation of the property, yields traces that reach the given state or that eventually make the transition fire. Other variants use mutations of models or properties to generate test suites.

Symbolic execution runs an (executable) model not with single input values but with *sets of input values* (e.g., [49, 50, 51]). These are represented as constraints. In this way, symbolic traces are generated: one symbolic trace represents many fully instantiated traces. The instantiation to concrete values must obviously be performed in order to get test cases for a SUT. Symbolic execution is guided by test case specifications. Often enough, these boil down to reachability statements as in the case of model checking. In other cases, test case specifications are given as explicit constraints, and the symbolic execution is guided by having to respect these constraints.

Deductive theorem proving can also be used for the generation of tests (e.g., [52, 53]), particularly with provers that support the generation of witness traces or counter-examples. One variant is similar to the use of model checkers where a theorem prover replaces the model checker. Most often, however, theorem provers are used to check the satisfiability of formulas that directly occur as guards of transitions in state-based models. A theorem prover can compute assignments for the variables that occur in the guards and that, in turn, give rise to values of the respective input and output signals. A sequence of such sets of signals then becomes the test case.

Constraint solving is useful for selecting data values from complex data domains, e.g., in combinatorial n-wise testing. It is also often used in conjunction with other methods such as symbolic execution, graph search algorithms, model-checking or theorem proving [54, 55] where specific relationships between variables in guards or conditions are expressed as constraints and efficiently solved by dedicated constraint solvers.

Test generation tools often use several techniques to complete the difficult task of automated test generation from a model. For example, theorem proving may be used to detect transition unreachability, while constraint solving is used in the same test generation engine for test data selection.

3.6. Test execution

The last dimension is concerned with test execution and the relative timing of test case generation and test execution.

Test execution is done in either *online* or *offline* from the test generation. Some tools (like Spec Explorer, described in the next section) support both approaches.

With online testing, the test generation algorithms can react to the actual outputs of the SUT. This is sometimes necessary if the SUT is non-deterministic, because the test generator can see which path the SUT has taken, and follow the same path in the model (Section 3.2). Offline testing means that test cases are generated strictly before they are run. Offline test generation from a non-deterministic model is more difficult, and involves creating test cases that are trees or graphs rather than sequences.

The advantages of offline testing, when applicable, are directly connected to the generation of a test repository. The generated tests can be managed and executed using existing test management tools, which means that fewer changes to the test process are required. One can generate a set of tests once, then execute it many times on the SUT (e.g., regression testing). Also, the test generation

DOI: 10.1002/stvr

and test execution can be performed on different machines or in different environments, as well as at different times. Test suites can be split and applied to many SUTs in parallel. It is also possible to perform a separate test minimisation pass over the generated test suite, to reduce the size of the test set. Moreover, testing real-time systems may be impossible if test generation is too time-consuming. Finally, if the test generation process is slower than test execution, then there are obvious advantages to doing the test generation phase just once.

Note that the generated test cases may be executed manually or automatically. Manual test execution means that a human tester executes each generated test case by interacting with the SUT, following the instructions in the test case, whereas automated test execution means that the generated test is already an executable test script of some form. However, this distinction between manual and automated test execution is not included in the the taxonomy because in practice, all approaches can support both manual and automatic execution of the generated tests (provided that suitable interfaces exist for automating the execution, which is not always the case, as shown by the example of testing "a car" by driving it).

For example, if each generated test case is just a sequence of keywords, it could be executed manually, or one can write an adaptor program that reads those keywords and executes them automatically. Automatic execution typically requires more work, to develop an adaptor program or library, but some of this overhead can be ameliorated if the adaptor code is reused for many different tests or for several different versions of the generated tests.

4. CLASSIFICATION OF TOOLS

This section classifies some typical model-based testing tools within the dimensions defined in Section 3. The purpose is to show the characteristics of those tools and the choices made for each dimension in order to target various application domains. This shows that the taxonomy is useful for discriminating between different model-based testing tools.

4.1. AETG

In *combinatorial testing* the issue is to reduce the large number of possible combinations of input variables to a few 'representative' ones. AETG (Automatic Efficient Test Generator [23]) is a model-based test input generator for combinatorial testing. To reduce the number of generated test inputs, it uses a pair-wise algorithm to ensure that all combinations of the data values for each *pair* of variables are tested. It also supports all-triples or all-quadruples testing. The oracle for each test input has to be provided manually. There are a large number of related tools dedicated to pair-wise testing (`www.pairwise.org`). A typical application domain for this approach is to test different configurations, for example device combinations or possible options for configuring some product.

Scope of the model: This is a typical input-only model, that is, a model of the environment. Pair-wise testing (and other n-way testing) uses a simple static model of the input data of the SUT, defining the domains of variables and any unauthorised combinations of values.
Model Characteristics: Models are untimed and discrete. The choice between determinism and non-determinism is not applicable, since AETG models do not provide expected output.
Modelling Paradigm: Models are expressed using the 'input domain' paradigm, which is a special case of the pre/post paradigm (data domains plus constraints).
Test Selection Criteria: This class of tools use data coverage criteria such as all-pairs coverage.
Technology: Generation of test inputs using n-way search algorithms.
Online/offline: AETG (and the other tools of the same category) generates tests for offline manual execution, or automated offline execution with manual development of the associated test scripts.

4.2. JUMBL

The *J Usage Model Builder Library* (JUMBL) [44] is an academic model-based statistical testing tool [41], developed at the University of Tennessee. JUMBL supports the development of statistical

DOI: 10.1002/stvr

usage based models using Markov chains, the analysis of models, and the generation of test cases. Test inputs are generated by traversing the usage model while respecting transition probabilities: the test cases with greatest probability are generated first. The usage model does not provide the expected response of the system. Similar tools include the Matelo system from ALL4TEC [56].

Scope of the model: The usage model represents the intended use of software, as defined by the specification, so is a model of the expected environment and is input data only.
Model Characteristics: Models are untimed and discrete. The choice between determinism and non-determinism is not relevant, since only test inputs are generated and SUT behaviour is not modelled.
Modelling Paradigm: JUMBL models are written in a transition-based notation for describing Markov chain usage models. A Markov chain usage model has a unique start state, a unique final state, a set of intermediate usage states, and transition arcs between states. The transition arcs are labelled by the corresponding event and the probability of occurrence. Transition probabilities are based on expected use of the SUT.
Test Selection Criteria: JUMBL uses random and statistical test selection criteria (based on the transition arc probability of the usage model).
Technology: Automated generation of test inputs using statistical search algorithms and the Markov model.
Online/offline: The generated test cases need to be translated into a script language of a test execution environment (or executed manually). The JUMBL primarily uses an offline approach, but also provides an API for relating the test execution results back to the model for statistical analysis.

4.3. Microsoft Spec Explorer

Spec Explorer [57] was developed within Microsoft Research during the last seven years and is used extensively within Microsoft on a daily basis. It has now been productized, and is planned to be released with Visual Studio 2010. It provides a model editing, composition, exploration and visualization environment within Visual Studio, and can generate offline .NET test suites or execute tests as they are generated (online). Other examples of commercial MBT tools that use behavioural test models are Qtronic from Conformiq [58] and CertifyIt from Smartesting [59].

Scope of the model: The input model of Spec Explorer is a SUT input-output model, which is typically composed from several simpler models. The model provides the oracle for each generated test case. Preconditions associated with the action methods of C# models can be used to model some environmental assumptions.
Model Characteristics: Models are untimed and discrete. Non-determinism is supported by distinguishing *controllable* actions from *observable* actions–the latter may be generated spontaneously by the SUT.
Modelling Paradigm: State-based models are written in C# (extended with preconditions), and a regular expression notation is used to specify history-based (trace-based) models/scenarios. The C# models can have internal state and parameterized methods with complex parameters. Multiple models written in these notations are composed to obtain the final SUT model.
Test Selection Criteria: Spec Explorer provides several strategies for managing the exploration of the model, including **data coverage** of parameter values and the state space (the state space can be restricted by several grouping and slicing techniques), and **structural model criteria** such as covering all transitions. A regular expression can also be used as an explicit **test case specification**–when it is composed with a general C# model of a SUT it can restrict the generated tests to focus on that scenario.
Technology: It uses algorithms similar to bounded model checking to explore the model and generate tests.
Online/offline: It supports both online and offline testing.

5. RELATED WORK

The last three decades have seen substantial research in the area of model-based testing. Broy et al. have provided a comprehensive overview of research in the field [60], see also the surveys of Dias-Neto et al. [3] and Hierons et al. [2]. An early focus of this research was conformance

DOI: 10.1002/stvr

testing between finite state machines; see Gargantini's review [61]. Binder's book concentrates on the idiosyncrasies of testing object-oriented software [62]. Research topics deal with the underlying algorithms, theory and technology of model-based testing. The focus of this article is more on the user perspective of model-based testing and evaluating and comparing different MBT approaches and associated tools. In the last few years, the field of model-based testing has moved from a research topic to an emerging practice in industry, with increasing commercial tool support.

Tools for test case generation have been surveyed by several authors. The 2002 survey by Alan Hartman [63] in the context of an EU-funded research project is now outdated. More recent surveys include that by Belinfante et al. [64] as well as that by Götz et al. [65]–the latter is a detailed study of nine commercial MBT tools, but is in German only. While rather comprehensive, these surveys were not based on an underlying taxonomy.

A recent systematic review of state-based MBT tools by Shafique and Labiche [66] gives a detailed classification of nine commercial and research MBT tools. The review covers only a subset of the taxonomy defined here, focussing on tools that use state-based models (i.e. Model Paradigm = Transition-Based[†]) to model the SUT behavior (i.e. Model Scope = Input-Output), and excluding MBT tools for embedded systems (i.e. limited to Model Characteristics = Discrete). The review does not discuss Test Generation Technology, nor whether the tools support Timed/Untimed models or Deterministic/Non-Deterministic models. The most interesting difference is that the review classifies the Test Selection Criteria into four groups: its *model-flow criteria* and *script-flow criteria* are both subsets of the Structural Model Coverage of this taxonomy, while its *data criteria* and *requirement criterion* correspond to Data Coverage and Requirements Coverage of the taxonomy respectively. The review does not discuss Random & Stochastic or Fault-Based test selection criteria (it would be useful to add these to the review), and classifies Test Case Specifications (ie. user-defined scenarios) as just another kind of *model-flow criteria*. In contrast, the taxonomy here views ad-hoc test case specifications as completely distinct from structural model coverage criteria, because test case specifications are user-defined, whereas structural model coverage criteria are used to automate the selection of tests. Thus ad-hoc test case specifications allow users a high degree of control over test generation, which is not possible using just structural model coverage criteria. The review also reports on several tool characteristics related to the convenience of using a given tool, such as whether models can be edited, checked and debugged within the tool or via a separate tool, and the degree of automation for generating various kinds of test scaffolding code (e.g., adapter code, oracle code, test stubs). So the review is complementary to the taxonomy in that it classifies several example tools, within a subset of the taxonomy, in great detail.

Dias-Neto et al. have performed a systematic review and classification of the MBT research literature, based on keyword searches of six digital libraries [67]. They found 599 papers and analyzed 271 of those papers from between 1990 and August 2009 that were available online and were about MBT. In those papers, they identified 219 different approaches to MBT. They classified the approaches according to 29 different attributes, such as whether the models used UML or not, whether the goal was functional or non-functional testing, the testing level (system/integration/unit/regression testing), the level of automation, and various other attributes about the model, the test generation process and the software development environment within which MBT was used. This review complements the taxonomy presented in this paper, because the review gives a very detailed view of many existing MBT approaches, while the taxonomy gives a higher-level way of classifying both existing and future MBT approaches. For example, the review lists 48 different MBT modeling notations [68], while the taxonomy groups these into seven modelling paradigms. Another difference is that while the review covers four of the taxonomy dimensions, it does not have any attributes that directly correspond to the *Model Scope* (input-only or input-output) or *On/Offline* dimensions. In some cases, it is possible to infer the model scope, based on the kind of the model that was used, but in general these two dimensions capture important information about a MBT approach that are missing from the review attributes.

[†]Eight of the tools in the review use FSM/EFSM models, while Spec Explorer starts with a Pre-Post model, then expands it into a FSM model.

DOI: 10.1002/stvr

Dias-Neto et al. also performed an email survey of 34 MBT researchers, asking them to rank 18 MBT-related attributes, to determine which are the most important attributes when classifying or selecting MBT approaches [69]. The highest-ranked attribute was the kind of model used, which is the first top-level dimension of the taxonomy in this paper, and the next two highest attributes were the test generation criteria and coverage, which correspond to the *Test Selection Criteria* dimension of the taxomony. The *On/Offline* dimension of the taxonomy was omitted from the survey, but is included in the taxonomy because it has a major impact on both the theory and practice of MBT: it has a theoretical impact on the kinds of SUT that can be tested (online is better for non-deterministic SUTs), and it has a strong practical impact on the integration of MBT into existing software testing processes (offline is easier to integrate, while online is more disruptive).

6. CONCLUSIONS

The idea of model-based testing is to use an explicit abstract model of a SUT and/or its environment to automatically derive tests for the SUT: the behaviour of the model of the SUT is interpreted as the intended behaviour of the SUT. This approach is particularly appealing because it assigns a threefold use to models: they are used to come to grips with precise requirements descriptions, they can be used as parts of specification documents, and they can be used for test case generation.

The emerging nature and increasing popularity of the field of model-based testing have led to a large body of publications. But there is currently a lack of a unifying conceptual framework, which makes it difficult to compare different approaches. The taxonomy of this paper provides the essential characteristics of the various mainstream approaches to model-based testing, both academic and industrial. The usefulness of the taxonomy has, in a deliberately exemplary manner, been demonstrated by classifying several MBT tools w.r.t. its dimensions.

Taxonomies have been proposed in several areas of computer science, such as runtime monitoring [70], mining of source code repositories [71] and software product lines [72]. Such taxonomies help to clarify the key issues of the field and show the possible alternatives and directions. They can be used to classify tools and to help users to see which approaches and tools fit their specific needs most closely. The development of a taxonomy for a given field requires that field to have a certain maturity, and sufficient experience with different approaches. It appears justified to assert that model-based testing has now reached a sufficient level of maturity for a taxonomy to be important and useful.

The technology of automated model-based test case generation has matured to the point where the large-scale deployment of this technology is underway. In terms of positive failure-detecting effectiveness, the body of evidence is rather large. In terms of cost effectiveness, there is room for further empirical investigations, including the analysis of the conditions (organisational, qualification of the team for example) that increase the effectiveness of the approach.

In addition to empirical studies, there are several research challenges, firstly in the definition of domain-specific test selection criteria that are empirically underpinned in terms of their fault detection power. Secondly, the tool support for building, versioning, and debugging models could be improved, and there is a need for more methodological guidance on how to build models for test generation. Thirdly, the question of model-based testing for non-functional requirements such as security or usability, as well as performance, is still an open issue. Finally, the performance of test generation and the improvement of test generation techniques is also an area of ongoing research, with the potential to further improve the scalability of current MBT approaches and tools.

REFERENCES

1. T. S. Chow, Testing software design modeled by finite-state machines, IEEE Trans. Softw. Eng. 4 (3) (1978) 178–187.
2. R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. Simons, S. Vilkomir, M. Woodward, H. Zedan, Using formal specifications to support testing, ACM Computing Surveys 41 (2) (2009) 9:1–9:76.

DOI: 10.1002/stvr

3. A. C. Dias Neto, R. Subramanyan, M. Vieira, G. H. Travassos, A survey on model-based testing approaches: a systematic review, in: WEASELTech '07: Proceedings of the 1st ACM Int. workshop on Empirical assessment of SW Eng. languages and technologies, ACM, New York, NY, USA, 2007, pp. 31–36.
4. M. Utting, B. Legeard, Commercial Model-Based Testing Tools, http://www.cs.waikato.ac.nz/research/mbt/Tools.pdf, last accessed November 2010 (2010).
5. C. Willcock, et al., An Introduction to TTCN-3, Wiley, 2005.
6. M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufman, 2007.
7. A. Dias-Neto, R. Subramanyan, M. Vieira, G. H. Travassos, F. Shull, Improving evidence about software technologies: A look at model-based testing, IEEE Software 25 (3) (2008) 10–13.
8. A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, T. Stauner, One evaluation of model-based testing and its automation, in: Proc. ICSE'05, 2005, pp. 392–401.
9. A. Paradkar, Case studies on fault detection effectiveness of model based testing generation techniques, in: Proc. ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST'05), Vol. 30, 2005, pp. 1–7.
10. M. Horstmann, W. Prenninger, M. El-Ramly, Case Studies, in: Broy et al. [60], pp. 439–461.
11. E. Bernard, B. Legeard, X. Luck, F. Peureux, Generation of test sequences from formal specifications: GSM 11.11 standard case-study, SW Practice and Experience 34 (10) (2004) 915 – 948.
12. M. Blackburn, R. Busser, A. Nauman, Why model-based test automation is different and what you should know to get started, in: Proc. Intl. Conf. on Practical Software Quality and Testing, 2004.
13. E. Farchi, A. Hartman, S. S. Pinter, Using a model-based test generator to test for standard conformance, IBM Systems Journal 41 (1) (2002) 89–110.
14. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, B. M. Horowitz, Model-based testing in practice, in: Proc. ICSE'99, 1999, pp. 285–294.
15. J. Jacky, M. Veanes, C. Campbell, W. Schulte, Model-Based Software Testing and Analysis with C#, Cambridge University Press, 2008.
16. A. Pretschner, J. Philipps, Methodological Issues in Model-Based Testing, in: Broy et al. [60], pp. 281–291.
17. W. Prenninger, A. Pretschner, Abstractions for Model-Based Testing, ENTCS 116 (2005) 59–71.
18. T. Koomen, L. van der Aalst, B. Broekman, M. Vroon, TMap Next, for result-driven testing, UTN Publishers, 2006.
19. D. Graham, E. V. Veenendaal, I. Evans, R. Black, Foundations of Software Testing: ISTQB Certification, Int. Thomson Business Pr, 2008.
20. J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, K. Scholl, Model-based test case generation for smart cards, in: Proc. 8th Intl. Workshop on Formal Meth. for Industrial Critical Syst., 2003, pp. 168–192.
21. K. Berkenkötter, R. Kirner, Real-Time and Hybrid Systems Testing, in: Broy et al. [60], pp. 355–387.
22. A. van Lamsweerde, Formal specification: a roadmap, in: Proc. ICSE'00, 2000, pp. 147–159.
23. D. Cohen, S. Dalal, M. Fredman, G. Patton, The AETG System: An approach to testing Based on Combinatorial Design, IEEE TSE 23 (7) (1997) 437–444.
24. M. Gaudel, P. LeGall, Testing data type implementations from algebraic specifications, in: Proc. Formal Methods and Testing, Springer LNCS 4949, 2008, pp. 209–239.
25. A. Aho, A. Dahbura, D. Lee, M. U. Uyar, An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours, IEEE Transactions on Communications 39 (11) (1991) 1604–1615.
26. D. Lee, M. Yannakakis, Principles and methods of testing finite state machines — A survey, Proceedings of the IEEE 84 (2) (1996) 1090–1126.
27. H. Zhu, P. Hall, J. May, Software Unit Test Coverage and Adequacy, ACM Computing Surveys 29 (4) (1997) 366–427.
28. S. Ntafos, A Comparison of Some Structural Testing Strategies, IEEE TSE 14 (6) (1988) 868–874.
29. P. Frankl, E. Weyuker, An Applicable Family of Data Flow Testing Criteria, IEEE TSE 14 (10) (1988) 1483–1498.
30. M. Grindal, J. Offutt, S. Andler, Combination testing strategies: A survey, Tech. Rep. ISE-TR-04-05, George Mason University (2004).
31. N. Kosmatov, B. Legeard, F. Peureux, M. Utting, Boundary coverage criteria for test generation from formal models, in: Proc. 15th Intl. Symp. on SW Reliability Engineering, 2004, pp. 139–150.
32. B. Beizer, Black-Box Testing : Techniques for Functional Testing of Software and Systems, Wiley, 1995.
33. D. Hamlet, R. Taylor, Partition Testing Does Not Inspire Confidence, IEEE TSE 16 (12) (1990) 1402–1411.
34. J. Duran, S. Ntafos, An Evaluation of Random Testing, IEEE TSE SE-10 (4) (1984) 438–444.
35. W. Gutjahr, Partition testing versus random testing: the influence of uncertainty, IEEE TSE 25 (5) (1999) 661–674.
36. V. Nair, D. James, W. Ehrlich, J. Zevallos, A Statistical Assessment of some Software Testing Strategies and Application of Experimental Design Techniques, Statistica Sinica 8 (1998) 165–184.
37. C. Gaston, D. Seifert, Evaluating Coverage-Based Testing, in: Broy et al. [60], pp. 293–322.
38. J. Wittevrongel, F. Maurer, Scentor: Scenario-based testing of e-business applications, in: WETICE '01: Proc. of the 10th IEEE Int. Workshop on Enabling Technologies, IEEE Comp. Society, Washington, DC, 2001, pp. 41–48.
39. W. T. Tsai, A. Saimi, L. Yu, R. Paul, Scenario-based object-oriented testing framework, QSIC 00 (2003) 410.
40. J. Julliand, P.-A. Masson, R. Tissot, Generating security tests in addition to functional tests, in: AST'08, 3rd Int. workshop on Automation of Software Test, ACM Press, Leipzig, Germany, 2008, pp. 41–44. URL http://doi.acm.org/10.1145/1370042.1370051
41. G. Walton, J. Poore, Generating transition probabilities to support model-based software testing, Software: Practice and Experience 30 (10) (2000) 1095–1106.
42. J. Musa, Software Reliability Engineering, AuthorHouse, 2nd ed., 2004.
43. J. Andrews, L. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments, in: Proc. ICSE'05, 2005, pp. 402–411.
44. S. Prowell, Jumbl: A tool for model-based statistical testing, in: Proc. HICSS'03, IEEE, 2003, p. 337.3.
45. M. Kwan, Graphic programming using odd and even points, Chinese Mathematics 1 (1962) 273–277.

DOI: 10.1002/stvr

46. P. McMinn, Search-based software test data generation: a survey, J. Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
47. A. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications, J. Software Testing, Verification and Reliability 13 (1) (2003) 25–53.
48. H. Hong, I. Lee, O. Sokolsky, H. Ural, A Temporal Logic Based Theory of Test Coverage and Generation, in: Proc. TACAS'02, 2002, pp. 327–341.
49. A. Pretschner, Classical search strategies for test case generation with Constraint Logic Programming, in: Proc. Formal Approaches to Testing of Software, 2001, pp. 47–60.
50. B. Marre, A. Arnould, Test Sequences Generation from LUSTRE Descriptions: GATEL, in: Proc. 15th IEEE Conf. on Auto. SW Eng., 2000, pp. 229–237.
51. S. Colin, B. Legeard, F. Peureux, Preamble computation in automated test case generation using Constraint Logic Programming, J. Software Testing, Verification and Reliability 14 (3) (2004) 213–235.
52. J. Dick, A. Faivre, Automating the generation and sequencing of test cases from model-based specifications, in: Proc. 1st Intl. Symp. of Formal Methods Europe, Vol. 670 of LNCS, 1993, pp. 268–284.
53. S. Helke, T. Neustupny, T. Santen, Automating test case generation from Z specifications with Isabelle, in: Proc. 10th Intl. Conf. of Z Users, Vol. 1212 of LNCS, 1997, pp. 52–71.
54. D. Clarke, T. Jron, V. Rusu, E. Zinovieva, Stg: a symbolic test generation tool, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Vol. LNCS 2280, Springer-Verlag, 2002, pp. 470–475. URL http://www.irisa.fr/vertecs/Publis/Ps/2002-TACAS.ps.gz
55. S. Colin, B. Legeard, F. Peureux, Preamble Computation in Automated Test Case Generation using Constraint Logic Programming, The Journal of Software Testing, Verification and Reliability 14 (3) (2004) 213–235.
56. All4Tec, web site at http://www.all4tec.net/index.php/All4tec/matelo-concept.html. (2010).
57. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillman, L. Nachmanson, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, Vol. 4949 of LNCS, Springer-Verlag, 2008, Ch. 2, pp. 39–76.
58. Conformiq, web site at http://www.conformiq.com/products.php. (2010).
59. Smartesting, web site at http://www.smartesting.com. (2010).
60. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (Eds.), Model-Based Testing of Reactive Systems, no. 3472 in LNCS, Springer-Verlag, 2005.
61. A. Gargantini, Conformance Testing, in: Broy et al. [60], pp. 87–111.
62. R. V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.
63. A. Hartman, Model-based test generation tools, Tech. rep., AGEDIS Project, available from http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf. (2002).
64. A. Belinfante, L. Frantzen, C. Schallhart, Tools for Test Case Generation, in: Broy et al. [60], pp. 391–438.
65. H. Götz, M. Nickolaus, T. Roner, K. Salomon, Modellbasiertes Testen, Heise Zeitschriften Verlag GmbH, 2009.
66. M. Shafique, Y. Labiche, A systematic review of model based testing tool support, Tech. Rep. SCE-10-04, Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada (May 2010).
67. A. C. Dias-Neto, G. H. Travassos, A picture from the model-based testing area: Concepts, techniques, and challenges, Advances in Computers 80 (2010) 45–120.
68. A. C. Dias-Neto, G. H. Travassos, R. Subramanyan, M. Vieira, Characterization of model-based software testing approaches, Tech. Rep. ES-713/07, PESC-COPPE/UFRJ, available at http://www.cos.ufrj.br/uploadfiles/1188491168.pdf (2007).
69. A. C. Dias Neto, G. H. Travassos, Surveying model based testing approaches characterization attributes, in: ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, ACM, New York, NY, USA, 2008, pp. 324–326.
70. N. Delgado, A. Q. Gates, S. Roach, A taxonomy and catalog of runtime software-fault monitoring tools, IEEE Transactions on Software Engineering 30 (12) (2004) 859–872.
71. H. Kagdi, M. Collard, J. Maletic, Towards a taxonomy of approaches for mining of source code repositories, SIGSOFT SW. Eng. Notes 30 (4) (2005) 1–5.
72. C. W. Krueger, Integration testing in software product line engineering: A model-based technique, in: FASE'07: Proceedings of Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2007, pp. 321–335.

DOI: 10.1002/stvr