A Systematic Review of Model Based Testing Tool Support

Muhammad Shafique, Yvan Labiche

Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa ON K1S5B6, Canada
mshafiqu@connect.carleton.ca, labiche@sce.carleton.ca

Abstract. Model-based testing (MBT) is about testing a software system by using a model of its behaviour. To benefit fully from MBT, automation support is required. This paper presents a systematic review of prominent MBT tool support where we focus on tools that rely on state-based models. The systematic review protocol precisely describes the scope of the search and the steps involved in tool selection. Precisely defined criteria are used to compare selected tools and comprise support for test coverage criteria, level of automation for various testing activities, and support for the construction of test scaffolding. The results of this review should be of interest to a wide range of stakeholders: software companies interested in selecting the most appropriate MBT tool for their needs; organizations willing to invest into creating MBT tool support; researchers interested in setting research directions.

Keywords: state-based testing, transition-based testing, systematic review.

1 Introduction

Software testing, that is evaluating software by observing its executions on actual valued inputs [1], is probably the most widely used verification and validation technique. Software testing techniques are usually characterized as being black-box (or functional) or white-box (or structural) depending on whether they rely solely on a specification of the software under test (SUT) or solely on its implementation [2]. Authors also introduce the notion of grey-box testing that mixes information from the specification and the implementation to derive tests [14].

Another term that is more and more used in textbooks and articles is model-based testing (MBT) [5, 7, 17]. Utting and Legeard recognize that this term has different acceptions and they focus on one which is the automation of the design of black-box tests [17]. In this paper we consider a slightly broader definition and consider model-based testing as the support of one or more (the more the merrier) software testing activities from a model of the SUT behaviour. These activities include: constructing abstract test cases (or test case specifications), constructing concrete (executable) test cases (sometimes referred to as test scripts), constructing a test oracle (i.e., a means that provides information about the correct, expected behavior of the SUT [2]. Depending on the level of abstraction of the model of the SUT behaviour, MBT can

be considered black-box or grey-box: an abstract representation of the SUT behaviour leads to black-box MBT, whereas a model that includes design information for instance leads to grey-box MBT. (Note that, although a control flow graph derived from the source code can be considered a model of the SUT, white box testing is not usually considered MBT.) During MBT, the model of the SUT can come from development phases of the SUT, for instance following standard model-driven development processes [6]—the intent is to leverage information (model) that is already available—or it can be devised specifically for testing purposes. The rationale for creating models specifically for MBT, although this may mean creating models in addition to existing analysis and design models (i.e., additional costs), it to create simpler, more concrete, more detailed, more focused (and often more ready to use for testing activities) models [17].

There is more and more interest in MBT since it promises early testing activities, hopefully early fault detection (e.g., [4]). However, to get full benefits from MBT, automation support is required, and the level of automation support of a specific MBT technique is likely to drive its adoption by practitioners.

When practitioners want to adopt MBT, they therefore seek existing modeling technologies and associated MBT tools. Similarly, companies willing to invest in creating an MBT tool would be interested in knowing what exists to be able to really compete. Last, researchers involved in MBT research (e.g., devising new MBT technology, evaluating and leveraging existing MBT technologies) would equally be interested in knowing what currently exists. Seeking such information is a difficult task since there is a plethora of modeling technologies that can support some kind of MBT activity (e.g., finite state machines and all its variations, model of input data domain, the Unified Modeling Language) and there is a plethora of existing MBT tools (either commercial, open source, or academic prototype), and those technologies and tools have varying capabilities and are therefore difficult to compare. There exist some resources that report on such tools and provide some kind of comparisons (e.g., [5, 17]). However, the selection procedure of the tools being evaluated is often not reported (we can assume selection is ad-hoc), and therefore the list of tools is likely not complete (enough), and comparison is often weak. (See section 6 for further details.)

In this paper we are interested in helping the above-mentioned stakeholders (i.e., practitioners, tool vendors, researchers) answer questions about the capabilities of existing MBT tools. Because of the many modeling languages, we focus on MBT tools where some kind of state-based modeling language is used to model the SUT behaviour. On the one hand, this is a pragmatic decision as such models seem to be used for many MBT activities (as suggested by the many tools we have found), and on the other hand, comparing capabilities of tools relying on widely varying modeling notations would probably be less interesting: many tool capabilities are likely directly related to the input (modeling) language.

Answering the abovementioned questions is however not an easy task. First we do not want to miss any interesting MBT tool. Second, we need precise (or as precise as possible) comparison criteria.

In this paper we therefore conduct a systematic review “à la” Kitchenham [11] of model state-based testing tools. This is not strictly speaking a systematic literature review (SLR) [7, 10, 11] since we are not interested in evidence related to the use of

MBT: we are not looking for experiments where MBT is evaluated. We are simply interested in reporting on and comparing existing tools. However, to do so, we adopt principles from SLRs since we want "to present a fair evaluation of MBT tools by using a trustworthy, rigorous, and auditable methodology." [11].

We therefore first describe the systematic review protocol we have followed (section 2). The protocol describes the scope of the study, the steps we followed to identify (select) existing MBT tools. We then precisely discuss the evaluation criteria we have devised to compare selected tools (section 3). We then discuss the tools we selected (section 4) and report on the results of the systematic review (section 5). This work complements existing resources and works that report on MBT tools, as discussed in section 6. Conclusions are drawn in section 7.

2 Systematic Review Protocol

A systematic review identifies, evaluates and interprets available material to answer a research question [11] by following a review protocol which is developed before the start of the review itself. Adopting guidelines for systematic literature reviews in software engineering [11], our review protocol precisely describes the scope and objectives of the review, the research question(s), the search process, inclusion/exclusion criteria of identified tools and the evaluation criteria. Our study was mainly about MBT tools, though we also used research literature as support material to better understand MBT, and more specifically state-based MBT. For this reason, we studied considerable research and technical material to (1) identify inclusion/exclusion criteria to select so-called primary studies (i.e., the studied MBT tools) and (2) identify criteria to compare selected MBT tools. These criteria are part of the systematic review protocol.

2.1 Scope, Objectives, and Research Questions

The **scope** of our systematic review is state-based MBT tools available in research and commercial domains. Unless specified, we simply use *MBT tool* in the rest of this paper to refer to *state-based MBT tool*.

The main **objective** of this systematic review is to provide guidelines to MBT practitioners to select the most appropriate tool for their needs. Our work shall also help research or private organizations willing to invest in creating an MBT tool. Last, results should help researchers identify possible gaps between current MBT practice (as supported by commercial tools) and MBT research.

To achieve these objectives, the **research questions** we are interested in are:

- What are commercial and research MBT tools?
- What are the capabilities of those tools in terms (for instance) of test coverage, automation level and test scaffolding construction?

2.2 Search Process

Our search process started with an electronic search that included eight electronic resources, specifically ACM Digital Library, IEEE Explorer, SpringerLink, Elsevier, InterScience Wiley, EI Engineering Village, and CiteseerX, as well as Google Scholar (as suggested in [11]). Since we were interested in commercial MBT tools, which are not necessarily described in academic, peer-reviewed venues, we completed this initial search in four different ways. First, we studied several books on MBT (e.g., [5, 17]), or online documents (reports, articles) we found using the electronic search. We also manually consulted a large MBT discussion group (1,300 plus members), specifically model-based-testing@yahoo, a mailing list for software test professionals interested in MBT), and private organization web sites (e.g., AGEDIS, D-MINT—see Appendix B for details). Third, we completed the search by contacting prominent researchers in the domain (e.g., Alan Hartman, Mark Utting, Bruno Legeard). Last, we performed an electronic search in a (non-)academic, non-IT venue, specifically in Business Source Complete (scholarly business database). (Note that to identify this search engine, and other sources not mentioned here, which are unusual for IT researchers, we received help from a Librarian at Carleton University.) The search ended in December 2009.

Our search process is summarized below:

1. Identify search strings to perform electronic searches: see Appendix A for details.
2. Study of selected paper abstracts to identify relevance with our work.
3. Collect tool user manuals, white papers, technical reports, books, case studies, video tutorials and demonstrations, presentation, product descriptions and customer feedbacks available in hard and soft formats on the vendor website and the Internet.
4. Scan list of paper references, authors and/or research groups and projects web sites to collect more information.
5. Initiate and participate in technical discussions online to improve understanding of MBT tools.
6. We repeated steps 1-5 to prepare a list of MBT tools and supporting data until no new tool could be added to the list.

2.3 Inclusion and Exclusion Criteria

A systematic search, especially one involving queries to databases often leads to a large number of documents since query strings have a broad enough scope to avoid missing any important document. Inclusion and exclusion criteria are therefore necessary to discard irrelevant documents and only keep relevant ones, referred to as primary studies [11]. To select primary studies accurately, inclusion/exclusion criteria are set with respect to research questions. We defined two inclusion criteria (used also for exclusion since a study that does not satisfy the inclusion criteria is excluded), one for selection of supporting material and the other for selection of state-based MBT tools. We included:

1. Research papers, reports, technical articles, white papers, manuals, presentation and online videos that are relevant to MBT;
2. All MBT tools that use (Extended) Finite State Machines (FSM/EFSM), UML state machines, Harel statecharts, Labeled Transition Systems (LTS), Abstract State Machines (ASM).

3 Systematic Review Criteria

To extract information from primary studies and then answer research questions, a data extraction procedure as precise and accurate as possible (to avoid bias and allow replications) needs to be defined [11]. Since our research questions relate to several MBT tooling aspects that we deemed important, we defined and used the following comparison criteria: Test Criteria (section 3.1), Support for Related Activities Criteria (section 3.2), Test Scaffolding Criteria (section 3.3) and Miscellaneous Criteria (section 3.4). An initial sub-set of those criteria had been established prior to conducting the evaluation of selected tools (primary studies), and was then refined given what we learnt about the tools: the identification of criteria was iterative.

Note that gathering information about primary studies (tools) was not always an easy task. There are several reasons for that. Academic and open source tools often come with outdated or incomplete information. Fortunately, they are often described in published papers where we can collect information. Commercial tools come with user manuals and case study descriptions which often do not provide enough technical information. Tool vendors do not publish research papers for proprietary reasons. We used technical articles, white papers, case study reports, online videos and dedicated online user forums to get information we deemed important in our evaluation process. We also contacted prominent researchers and tool vendors to get unpublished information. We made every effort to collect accurate and precise information, but we were limited by what is explicitly available: e.g., we believe some tools provide additional functionalities such as a test criterion, and in some cases we reckon what that criterion is likely to be; however we decided not to report on what is uncertain.

3.1 Test Criteria

The comparison criteria referred to as *test criteria* measure the use of model-based test criteria for the construction of tests in MBT tools as well as criteria to measure coverage: distinction between adequacy criteria and coverage criteria. There is a wide variety of terms used in primary studies to refer to (sometimes the same) criteria. We tried to recognize similarities, synonyms (similarly to [1]) and used MBT tools terminology for uncommon criteria. We divide these criteria into four groups:

- *Model-flow criteria* refer to state, transition, transition-pair, sneak path, path, parallel-transition, and scenario criteria. These are adequacy criteria, i.e., they are used to build test cases. The first five are well-known criteria [1, 3]. Parallel-transition refers to exercising pairs of transitions from parts of the state model that execute in parallel (concurrently). Scenario is to ensure that sequences of states or transitions defined by the user are exercised by tests.

- *Script-flow criteria* refer to interface (function), statement, path, decision/branch, condition, modified-condition/decision and atomic-condition. Some MBT tools provide a mechanism to specify the SUT behavior further than simply with the state machine, which is then used to create drivers executing tests. Some tools use a scripting language; others use pre and post conditions for specifying function behaviour. These criteria refer to exercising parts of those specifications. They are well-known to the testing community [1] and are either used as adequacy criteria or coverage criteria in MBT tools. Note also that the last four can also apply to guard conditions in transitions (i.e., when a guard is a complex Boolean expression). They are however listed in script-flow to simplify the discussions.
- *Data criteria* are one-value, all-values, boundary-values, and pair-wise values, and refer to the selection of input values when creating concrete test cases from abstract test cases. Depending on the selected criterion, one or more concrete test case is created for every single abstract test case: one-value selects only one value for test inputs (one concrete test case is created for each abstract test case); all-values selects every possible values, which is often too expensive (if even possible), and creates as many concrete test cases as identified values for each abstract test case. Boundary values relies on boundaries of equivalence classes and more than one concrete test case is created for each abstract test case. These are adequacy criteria.
- *Requirement criterion* relies on traceability links between requirements and model elements (e.g., states, transitions) and is to create tests that exercise model elements linked to selected requirements.

For each group or coverage criterion, tool data will allow us to answer whether the tool supports the criterion (Y), whether the tool does not support the criterion (N), or whether the criterion is not relevant to the tool (N/A). Analysis will also tell us whether the criterion is an adequacy (AC) or a coverage (CC) criterion. Supporting a criterion in this context means being able to build test cases that satisfy the (adequacy) criterion or being able to indicate the level of coverage of the (coverage) criterion of a set of test cases.

3.2 Support for Related Activities Criteria

Beside the constructions of test cases from a model (section 3.1) and the construction of test scaffolding (section 3.3), MBT tools support a number of other activities that facilitate the integration of MBT activities in the larger process of software development. The related activities that we have recognized as already supported or that we consider important include model creation, sub-modeling, model verification, model debugging, and requirements traceability.

Model creation refers to the ability to create the test model with the MBT tool itself (through a GUI), or import the test model created by a third party tool. *Sub-modeling* refers to the ability to decompose the model into parts (i.e., sub-models) to reduce complexity. *Model verification* refers to the ability to check properties of the model (e.g., identify unreachable states or transitions) before test cases are generated. *Test case debugging* is related to debug test case to known its failure reasons: it is a

facility to help the user correct the model or requirement(s) or adapter code or the SUT. *Requirements traceability* refers to the ability to link requirements—often created by a third party tool (e.g., IBM DOORS)—to parts of the test model (e.g., a transition, a path), and therefore to test cases. Traceability provides three main benefits: one can see which test cases exercise which requirements; one can see which requirements are still to be tested; requirements linked to a failed test case can easily be identified.

These activities can either be directly supported by the MBT tool or they can be provided by a third party tool. In the former case, we are interested in the level of automation support of the activity whereas in the latter case, we are interested in seamless integration between the MBT tool and the third party tool.

These activities can be fully-supported (automated) by the MBT tool (F)—meaning that besides possibly providing some configuration data, the user can simply click to trigger the activity, or supported by a third party tool (TP), in which case the level of automation is that of the third party tool (reporting on third party tools is out of the scope of our study). In case a third party tool is used, the level of integration between this tool and the MBT tool can be seamless (S), easy (E) or difficult (D): seamless integration means the GUIs of the tools are related to one another; easy integration means for instance that a file has to be exported in one tool and imported into another; the integration is difficult if transferring data from one tool to the other is completely manual. Alternatively, these activities may not be supported at all (N), or they may be partially supported (P)—the user needs to be actively involved in conducting the activity.

3.3 Test Scaffolding Criteria

Test scaffolding usually refers to code developed to facilitate testing and includes test drivers (i.e., pieces of code to run tests), test stubs (e.g., pieces of code to simulate execution environment), and test oracles (i.e., pieces of code providing fail/pass verdicts) [15]. Since test scaffolding can facilitate online and offline testing, these two notions are also discussed in this section.

A test driver, also called test adapter, is a (ideally small) piece of code, (ideally) automatically generated, which is used to concretize and execute a test case (or test cases) on the SUT. The *test adapter criterion* measures the level of support provided by the MBT tool towards the creation of a test adapter. A tool can provide full support (F), in which case the adapter is automatically and entirely generated by the tool. Support can be partial (P): e.g., the tool automatically creates the skeleton of the adapter, for instance under the form of function signatures, and the user needs to fill the gaps (e.g., provide the body of the functions). Alternatively, a tool may provide no support (N). Support can also come from a third party tool (TP). Similarly, the *test stub criterion* and the *oracle criterion* measure the level of support provided by the MBT tool towards the creation of a test stub and oracle, respectively. Note that the oracle is often embedded inside the driver.

The offline criterion and online criterion measure the support for offline and online testing, respectively. A tool can support them (Y) or not (N). Offline testing means

that test cases are first generated and can then be executed, while online testing means that the test generator can react to how the SUT behaves.

3.4 Miscellaneous Criteria

These criteria provide additional information about tools to further help their selection and adoption. Although many aspects could be considered (e.g., licensing cost, customer support), we selected the following criteria since they seemed interesting: modeling notation (e.g., FSM, UML state machine), tool category (e.g., commercial, open source), and programming language of the SUT (e.g., .NET, Java, any).

4 Selected Tools

Following the search procedure detailed in section 2.2, we identified more than 300 documents (papers, research reports online resources, and technical manuals) and 27 model based testing tools. After applying inclusion/exclusion criterion 1 (section 2.3), we selected 78 documents. Using inclusion/exclusion criterion 2, we selected 12 tools as primary studies.

Three of those MBT tools, which are probably known to the reader, are however not discussed in this paper. These are Reactis (http://www.reactive-systems.com/), T-VEC Simulink Tester (http://www.t-vec.com/solutions/simulink.php), and MathWorks SystemTest (http://www.mathworks.com/products/systemtest/). These tools are not included in our review primarily because they target a specific family of systems (not necessarily software systems), specifically control, embedded systems which have to react to continuous inputs. As a result, they provide very specific verification and validation functionalities (e.g., model execution and simulation, code generation), which do not necessarily overlap with the other tools' functionalities. Indeed, when using our comparison criteria we often found that these Simulink/Stateflow chart based tools had capabilities that the others did not have, and vice versa. Therefore, we believe these tools should be compared with each other separately, likely according to dedicated comparison criteria.

Due to space constraints, we refer the reader to Appendix B for concise descriptions of the other nine tools, which are:

1. GOTCHA-TCBeans, generates test cases from FSM (www.haifa.ibm.com/projects/verification/gtcb/index.html).
2. mbt, generates test cases from (E)FSM (mbt.tigris.org).
3. MOTES, generates test cases from EFSM (www.elvior.ee).
4. TestOptimal, generates test cases from FSM (testoptimal.com).
5. AGEDIS, generates test cases from AML, a subset of UML (www.agedis.de/).
6. ParTeG, generates test cases from UML class and state machine diagrams (parteg.sourceforge.net).
7. Qtronic, generates test cases from UML state machine and scripting language (www.conformiq.com).
8. Test Designer, generates test cases from UML class, state machine and object diagram (www.smartesting.com).

9. Spec Explorer 2010, simply referred to as Spec Explorer in the rest of this paper, generates test cases from FSM and ASM (msdn.microsoft.com/en-us/devlabs/ee692301.aspx).

Fifteen other tools were not selected for review after applying inclusion/exclusion criterion 1 (section 2.3): ATD-Automated Test Designer, MaTelo, GATel, TOA, NModel, TGV, STG, TorX, exprecco, JUMBL, Uppaal TRON, PrUDE, TDE/UML, TestEra, and Escalator.

5 Results

In this section, we present the results of our systematic review. We use the notation described earlier in section 3.

5.1 Comparison 1: Test Criteria

Table 1 shows the adequacy criteria supported by the selected tools. All tools, except GOTCHA-TCBeans, fully support transitions coverage. Although this criterion is not explicitly supported (as described in GOTCHA-TCBeans documentation we used), the tool allows the user to define start and end states with forbidden states and transitions to help the tool find paths, thereby exercising some states and transitions. Spec Explorer does not support state criterion. MOTES, Qtronic and Test Designer support transition pair coverage. No tool except Qtronic supports sneak paths, all paths and parallel transitions coverage. MOTES and TestOptimal allow testers to create arbitrary scenarios (with order of states and transitions) by indicating different states. GOTCHA-TCBeans allows the user to create test scenarios using forbidden states and transitions but it does not guarantee to follow specific sub-paths. Similarly mbt and AGEDIS support limited coverage of states and transition (scenario) but do not allow testers to specify the order. We were not able to find some information for TestDesigner.

Table 1. State-based AC (adequacy criteria) comparison.

Tool Name	States	Transitions	Transition Pair	Sneak Paths	All Paths	Parallel Transitions	All Scenarios
GOTCHA-TCBeans	P	P	N	N	N	N/A	P
mbt	Y	Y	N	N	N	N/A	P
MOTES	Y	Y	Y	N	N	N/A	Y
TestOptimal	Y	Y	N	N	N	N/A	Y
AGEDIS	Y	Y	N	N	N	N	P
ParTeG	Y	Y	N	N	N	N	N
Qtronic	Y	Y	Y	Y	Y	Y	N
Test Designer	Y	Y	Y	?	?	N	?
Spec Explorer	N	Y	N	N	N	N/A	N/A

Table 2 shows script flow comparison. Interface/Function criterion refers to the coverage of those methods which are called on the SUT or present in a test model class. Interface/Function coverage is not applicable to mbt and MOTES because they

do not have scripting languages. AGEDIS, Qtronic and Test Designer provide mechanisms to further specify functions' baheviour to create test cases. They also inform whether the specified functions are covered or not (which is CC). Spec Explore covers the functions (actions) by generating transitions in FSM/ASM graph (which is AC). GOTCHA-TCBeans, TestOptimal and ParTeG do not inform about the coverage of functions or use that information for creating test cases.

There is a notion of statement and control flow structure in scripting languages GDL (GOTCHA-TCBeans), mScript (TestOptimal) and IF (AGEDIS) but, although, we believe this information is used to derive test cases, this is not clearly stated and tools do not report on how many statements and decisions are covered by test cases. In case of mbt, there is no notion of function, the notion of statement is only valid for the action part of the transition (the action part of transition can contain more than one actions/statements). Covering transitions ensures covering those statements and there is no notion of control flow in the action part of transition. mbt does not use any condition-related criterion to create test cases, similarly to MOTES. Since the test model of Spec Explorer (transformation of the user input model into a model that basically contains no guard, no action) only the Interface/Function criterion is applicable.

There is no scripting language in ParTeG and Test Designer so there is no notion of statement. Functions are class operations in the class diagram specified with OCL pre/post-conditions. Both these tools use condition-related criteria for guard conditions[1] (except atomic condition) but they do not use them for OCL contracts.

Qtronic supports all these criteria for model elements (guards) as well as in QML scripts, except for MC/DC.

Table 2. Script flow (adequacy and coverage) comparison.

Tool Name	Interface/ Function	Statement		Decision or Branch		Condition		MC/DC		Atomic condition	
		AC	CC	AC	CC	AC	CC	AC	CC	AC	CC
GOTCHA-TCBeans	N	N	N	N	N	N	N	N	N	N	N
mbt	N/A	N/A	N/A	N	N/A	N	N/A	N	N/A	N	N/A
MOTES	N/A	N/A	N/A	N	N/A	N	N/A	N	N/A	N	N/A
TestOptimal	N	N	N	N	N	N	N	N	N	N	N
AGEDIS	Y (CC)	N	N	N	N	N	N	N	N	N	N
ParTeG	N	N/A	N/A	Y	N	Y	N	Y	N	N	N
Qtronic	Y (CC)	Y	Y	Y	Y	Y	Y	N	N	Y	Y
Test Designer	Y (CC)	N/A	N/A	Y	N	Y	N	Y	N	N	N
Spec Explorer	Y (AC)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 3 shows that only Qtronic, Test Designer[1] and Spec Explorer generate test cases for data (adequacy) criteria. The latter two support One value and All values but their documentation does not recommend their use: the former is probably too simple (not effective at detecting faults) and the latter too expensive. Both these tools also support Boundary value and Pair-wise coverage. TestOptimal and Qtronic support Pair-wise coverage. Qtronic supports Boudary value but not the other three. Note that Spec Explorer does not provide GUI support to select criteria. The tester needs to use different data partition patterns in the input model to use these coverage criteria.

[1] Information about Test Designer was obtained through personal correspondence with Bruno Legeard.

Although AGEDIS generates parameterized test cases, which could be used to pass different input values through a test adapter, the tool itself does not support this. With GOTCHA-TCBeans, it is up to the tester creating the adapter to manually select input values according to any of these data criteria. Similarly, MOTES, mbt and TestOptimal rely on the user to provide test data. ParTeG' documentation suggests it supports Boundary Value, but this is not precisely documented.

Table 3. Data coverage and requirements coverage comparisons.

Tool Name	One value	All values	Boundary value	Pair-wise	Requirements
GOTCHA-TCBeans	N	N	N	N	N
mbt	N	N	N	N	Y (CC)
MOTES	N	N	N	N	N
TestOptimal	N	N	N	Y	N
AGEDIS	N	N	N	N	N
ParTeG	N	N	?	N	N
Qtronic	N	N	Y	N	Y
Test Designer[1]	Y	Y	Y	Y	Y
Spec Explorer	Y	Y	Y	Y	Y (CC)

Table 3 also indicates that four tools support requirements criterion, either coverage only or adequacy and coverage. This is mainly achieved by defining unique identifiers (IDs) for requirements and then attaching these IDs to test model transitions/states.

5.2 Comparison 2: Support for Related Activities Criteria

Table 4 indicates that most of MBT tools rely on third-party (TP) tools to create models. Only TestOptimal and Qtronic provide full support for model creation. AGEDIS, Test Designer and Spec Explorer provide seamless (S) integration with the third party tool while the user of GOTCHA-TCBeans, mbt, MOTES and ParTeG has to import models created externally. Note that GOTCHA-TCBeans and MOTES require input scripts, created using standard text editors. Spec Explorer extends Visual Studio to create the model. These clearly indicates that instead of investing time and money in creating such a support, most tool vendors prefer to use existing environments.

Four tools support model verification. Most of the tools which do not execute test cases (F in Table 4, Part II) do not support test case debugging. ParTeG uses Eclipse while Spec Explorer uses Visual Studio for this purpose. mbt, TestOptimal, Test Designer and Spec Explorer support sub modeling.

Only Test Designer fully supports requirements traceability thanks to its integration with a requirement engineering tool, namely DOORS (Part II). mbt, Qtronic and Spec Explorer support requirements coverage but only partially support traceability (no requirement engineering tool). Similarly to modeling, most MBT tools rely on third party tools for test case execution (with the exception of GOTCHA-TCBeans, AGEDIS, and Spec Explorer). All the tools generate test cases.

Table 4. Automation coverage comparison

Part I

Tool Name	Model Creation	Model Verification	Test Case Debugging	Sub-modeling
GOTCHA-TCBeans	TP-D	N	F	N
mbt	TP-E	N	N/A	TP-E
MOTES	TP-D	N	N/A	N
TestOptimal	F	F	N	F
AGEDIS	TP-S	N	F	N
ParTeG	TP-E	N	TP-S	N
Qtronic	F	F	N/A	N
Test Designer	TP-S	F	N/A	TP-S
Spec Explorer	TP-S	F	TP-S	F

Part II

Tool Name	Test Case Generation	Test Case Execution	Requirements Traceability
GOTCHA-TCBeans		F	N
mbt		TP-D	P
MOTES		TP-E	N
TestOptimal		TP-S	N
AGEDIS	F	F	N
ParTeG		TP-S	N
Qtronic		TP-E	P
Test Designer		TP-S	TP-S
Spec Explorer		F	P

5.3 Comparison 3: Test Scaffolding Criteria

A small amount of support is provided for adapter creation (Table 5). When there is support, it is partial (P): only a skeleton of functions is created automatically. Only ParTeG generates complete test adapters but this is a prototype tool and its approach might not be extendable to test large, complex models. Test Designer seamlessly integrates with a third party tool for that. Similarly, oracle creation is partially supported, and the same comments apply to ParTeG and Test Designer. No tool is able to generate stubs.

mbt partially supports online testing because it prompts the tester for input values during the test. MOTES and TestOptimal use third party tools for offline and online testing. Qtronic does not support offline testing. Online testing can be done by linking Qtronic with a test execution engine through a DLL plug-in. Test Designer and Spec Explorer do not support online testing.

5.4 Comparison 4: Miscellaneous Criteria

Miscellaneous criteria appear in Table 6. Four of the tools use (E)FSM, four use UML (class diagram, or/and object diagram, and state machine). Spec Explorer generates test cases from FSM/ASM. Note that all tools except GOTCH-TCBeans and Test Designer are available for evaluation. mbt, ParTeG and AGEDIS are free. Spec Explorer is free but the user has to buy Visual Studio to use it. All these tools can be

Table 5. Test scaffolding criteria comparison.

Tool Name	Adapter Creation	Oracle Automation	Stub Creation	Online Testing	Offline Testing
GOTCHA-TCBeans	P	P	N	F	F
mbt	P	P	N	P	TP-E
MOTES	N	P	N/A	TP-E	TP-E
TestOptimal	P	P	N	TP-S	TP-S
AGEDIS	P	P	N	N	F
ParTeG	F	F	N	N	TP-S
Qtronic	N	P	N/A	TP-S	TP-E
Test Designer	TP-S-P	TP-S-P	?	N	TP-S
Spec Explorer	P	P	N	N	F

used to test all types of software applications. Most of the tools do not assume a specific target plateform programming language: only GOTCHA-TCBeans, AGEDIS and ParTeG do.

6 Related Work

Most of the primary studies we have identified using our systematic search are conference and journal papers reporting on MBT academic works, sometimes reporting on some MBT prototype tools. These were not of interest in our work since we wanted to focus on commercial, or at least ready-to-use tools.

A systematic review of academic initiatives in MBT has been published in 2007 [13]. Its focus therefore differs from ours (academic vs. industry). Additionally, the comparison criteria used are different. They considered the types of models (including non-state-based models), the level of tool support and automation (although the analysis is more coarse grained than ours), the test criteria supported, intermediate models, and the complexity. It is also worth noting that contrary to a systematic literature review [11], the authors where not interested in evaluating current empirical evidence. Rather, one of the main outcomes was the identification of the lack of empirical evidence in MBT. In that sense, the authors performed a systematic mapping study [11], i.e., "a broad review of primary studies to identify what evidence is available"[11], rather than a systematic literature review.

Surveys of some MBT tools are discussed in [4, 16]. Again, these are not systematic reviews since no systematic procedure to identify primary studies (relevant tools) is described: and indeed some of our tools are not discussed in [4, 16]. Also, the studies do not focus on state-based MBT tools and comparison criteria are slightly different and complement ours: for instance they do not study support for test scaffolding. Additionally, comparison between tools is succinct. Other MBT tools descriptions and comparisons can be found [5, 9, 17, 18] (other sources are also available in Appendix B). Once again, these studies differ from ours in several ways: the search of MBT tools does not seem to be systematic, the comparison criteria are either too succinct or complement ours.

To summarize, a number of resources are available to increase one's understanding of existing MBT tool support. They all differ from our work in one way or another, as explained above. We see all these initiatives, including ours, as complementary.

Table 6. Miscellaneous criteria comparison.

Tool Name	Model Type	Category	Software Domain	Target Platform
GOTCHA-TCBeans	FSM	IBM Internal	All	C/C++, Java
mbt	FSM/EFSM	Open source	All	General
MOTES	EFSM	Research	All	General
TestOptimal	FSM	Commercial	All	General
AGEDIS	UML(AML)	Research	All	C/C++, Java
ParTeG	UML	Research	All	Java
Qtronic	UML	Commercial	All	General
Test Designer	UML	Commercial	All	General
Spec Explorer	FSM/ASM	Commercial	All	General

7 Conclusion

Model based testing (MBT) which, in this paper, refers to model-based testing as the support of one or more (the more the merrier) software testing activities from a model of the behaviour of the system under test, is growing in popularity in the industry [17]. At the same time, there is a large amount of MBT techniques that have been proposed in the literature (see for instance the review in [13]). Our experience with MBT shows a gap between what MBT tools support and research on MBT, i.e., a large part of MBT research does not (seem to) translate into MBT tool support. To better understand this gap, a first step has been to perform a systematic review of existing MBT tools, focusing on tools for state-based testing. Our systematic procedure (adapted from the literature on systematic literature surveys [11]) identified 27 tools and we compared nine of them. Our comparison criteria include model-based and script based adequacy/coverage criteria, support for related testing activities (e.g., creating the test model), and support for the construction of the test scaffolding (driver, stub, oracle). Results show that support for those comparison criteria varies a lot from MBT tool to MBT tool, except for simple criteria such as state/transition adequacy criteria, test case creation. One area where there is room for a lot of improvements is, we believe, support for test scaffolding, as well as support for test data selection criteria.

The main contributions of this paper are three-fold. First, it is the first time principles of systematic literature reviews are applied to study state-based MBT tools. Second, the systematic procedure allowed us to precisely define comparison criteria, which nicely complement what currently exists in available documentation that compares such tools. Note that the fact that our protocol was systematic should allow replications, extensions. Third, results precisely indicate how tools compare and which criteria they mostly fail to satisfy.

Our future work will include extending our study to other tools (such as the ones we omitted in this study), and extending the set of comparison criteria.

Acknowledgements: We would like to thank all the individuals who helped us gather information on MBT tools, such as (non-exhaustive list) members of the model-based-testing@yahoo group, Alan Hartman, Bruno Legeard, and Robert Smith (Librarian). Yvan Labiche is supported by NSERC.

References

[1] Ammann P. and Offutt J., *Introduction to Software Testing*, Cambridge University Press, 2008.
[2] Beizer B., *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Edition, 1990.
[3] Binder R. V., *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Object Technology, Addison-Wesley, 1999.
[4] Boberg J., "Early Fault Detection with Model-Based Testing," *Proc. ACM SIGPLAN workshop on ERLANG*, pp. 09-20, 2008.
[5] Broy M., Jonsson B., Katoen J.-P., Leucker M. and Pretschner A., Model-Based Testing of Reactive Systems, *Lecture Notes in Computer Science*, vol. 3472, 2005.
[6] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2nd Edition, 2004.
[7] Dyba T., Kitchenham B. and Jorgensen M., "Evidence-based software engineering for parishioners," *IEEE Software*, vol. vol. 22 (1), pp. 58-65, 2005.
[8] Hartman A. and Nagin K., "The AGEDIS Tools for Model-Based Testing," Proc. International Symposium on Software Testing and Analysis, pp. 129-132, 2004.
[9] Huima A., Invited Talk: Implementing Conformiq Qtronic, 2007
[10] Kitchenham B., Dyba T. and Jorgensen M., "Evidence-based software engineering," *Proc. ACM/ International Conference on Software Engineering*, pp. 273-281, 2004.
[11] Kitchenham B. A., "Guidelines for Performing Systematic Literature Reviews in Software Engineering,", 2007.
[12] Kull A., PhD Thesis: Model Based Testing of Reactive Systems, *Thesis*, Tallinn University of Technology, Faculty of Information Technology, 2009
[13] Neto A., Subramanyam R., Vieira M. and Travassors G., "A Survey on Model-Based Testing Approaches: A Systematic Review," *Proc. IEEE International Conference on Automated Software Engineering*, pp. 31-36, 2007.
[14] Patton R., Software Testing, *SAMS*, 2nd Edition, 2005.
[15] Pezze M. and Young M., *Softaware Testing and Analysis*, Wiley, 2008.
[16] Saifan A. and Dingel J., "Model-Based Testing of Distributed Systems," *School of Computing Queen's University Canada*, 2008.
[17] Utting M. and Legeard B., *Practical Model-Based Testing: A Tools Approach*, Morgan Kauffmann, 2007.

Appendix A Search String

The iterative search strategy to collect tools for primary study and support material is described in section 2.2. The importance of relevant and meaningful search strings is vital in systematic review as reviewers would not want to miss important literature (and tools) to answer research question.
Kitchenham [11] identifies six groups of terms to break down research question into individual facets to facilitate search process. They are population, intervention, comparison, outcomes, context and study design. Based on the research question described in section 2.1, we selected two relevant groups of terms which are population terms and intervention terms, and then further split intervention terms into two groups, Intervention Terms1 and Intervention Temrms2. In first phase of search process, we constructed sophisticated meaningful strings from Intervention Terms1 with boolean AND and OR operators. We simply concatenated Intervention Terms1 where logical conjunction and disjunction were not supported.
After considerable search iterations with resources mentioned in section 2.2, tools names (Intervention Terms2) were identified. In second phase of search process, we constructed meaningful search strings from Intervention Terms1 and Intervention Terms2 to collect tool specific material in form of papers, technical reports, tool manuals, case studies, presentations, online demos, video lectures and online discussions.

A.1 Population Terms

State/transition model based testing tools.

A.2 Intervention Terms1

Model based, testing, tool, tools, software, framework, frameworks, application, applications, unified modeling language, UML,, diagram, sequence, activity, class, state machine, collaboration, component, statechart, finite state machine, FSM, extended, EFSM, labeled transition system, LTS, input, output, IOLTS, Simulink/Stateflow chart, Simulink, Stateflow, model driven, coverage, criteria, analysis, timed automata, abstract, ASM, statemate, using, of, for and with.

A.3 Intervention Terms2

Qtronic, Reactics, Spec Explorer, TDE/UML, , Smartesting, Conformiq, D-MINT, Test Designer, T-VEC test suites, MaTelo, ATD-Automated Test Designer, errfix, GATel, mbt.tigris.org, ModelJUnit, NModel, ParTeG, Simulink Tester, Simulink/Stateflow chart tester, TestOptimal, TVG, Time Partition Test, TorX,

expecco, PrUDE, KeY, JUMBL, CTesk, Escalator, MOTES, AGEDIS, GOTCHA-TCBeans, TOA, Uppaal TRON, TestEra, exprecco and AsmL.

Appendix B Selected Tools

This section provides concise technical details of tools which we selected for analysis (sections 4 and 5).

B.1 GOTCHA-TCBeans

In GOTCHA-TCBeans, GOTCHA generates test cases from an FSM model while TCBeans provides Java classes to concretize and execute test cases on the SUT. The FSM test model is written in GDL (GOTCHA Definition Language) which is an extension of MDL (Murphi description language[2]). It specifies state variables (including types) and states, functions to be used in the model, stimuli of the FSM and expected results, as well as constraints to avoid the state space explosion problem during test case generation. The user can specify test scenarios. GOTCHA generates test cases in XML format for online and offline testing. Test adapters are written using TCBeans classes and a translation table in XML (or TCBeans classes) to map calls to methods of the SUT. Test cases can either be executed by an existing test execution engine with A2C (abstract to concrete) test translator TCBeans classes or without test execution engine by using A2E (abstract to executable) TCBeans classes. Test execution traces can be viewed in a TCBeans browser.

B.2 mbt

mbt is open source and does not have a GUI (command prompt only). mbt imports an FSM/EFSM model, possibly composed of sub-models (to handle complexity), in GraphML[3] format, for instance created with yED[4]. mbt provides requirements traceability. mbt provides different algorithms to generate test cases. It generates skeleton code for test adapters using a default or user provided code template. Placeholders are available to the user to describe the adapter behavior, including the oracle. mbt supports both online and offline testing. However, online testing is only partially automated because the tester has to provide input values when test sequences are run. mbt provides options for partial states, transitions and requirements coverage. It also generates test sequence to cover only specific states, transitions and requirements, as specified by the user. mbt does not support test case execution.

[2] http://verify.stanford.edu/dill/murphi.html

[3] http://graphml.graphdrawing.org/

[4] http://www.yworks.com

B.3 MOTES

MOTES is an Eclipse plug-in. It uses EFSMs to generate test cases. MOTES uses third party tools such as Poseidon[5] or Artisan Studio[6] for model creation: It imports test model in XMI format. MOTES requires three input files to generate test cases in TTCN-3: the test model in XMI format; test data description in TTCN-3; configuration data for EFSM states and input/output ports (these describe the types of data and the calls that can be sent to the SUT). Generated TTCN-3 test cases can then be executed by a third party (TTCN-3 compliant) tool (such as MessageMagic[7]). MOTES relies on Uppaal CORA[8] and MTL (MOTES transition language) to generate test cases [12]. MOTES supports online and offline testing. This tool is used as extensive case study in D-MINT project.

B.4 TestOptimal

TestOptimal is a web based client server tool that tests desktop and multitier enterprise applications. A TestOptimal model is an FSM, created interactively while analyzing the web site being tested. It can also be imported in GraphML[9], XMI[10] and GraphXML[11] formats. TestOptimal provides model validation, simulation and debugging support. It provides an XML based scripting language called mScript to connect (adapter/driver) the model to the SUT. A tester can test do scenario testing using mCase. TestOptimal provides multiple algorithms to generate test cases and supports online and offline testing. It can be used for stress, load and regression testing. TestOptimal automatically generates test adapter class skeleton where a tester can add function logic to run generated test cases.

B.5 AGEDIS

AGEDIS is a test suite for model based testing of component based distributed systems. It includes a modeling tool (Objecteeing[12] UML Modeler), a test suite editor (Spy editor[13]) and browser, a test case simulation and debugging tool, a test coverage analysis tool, a defect analysis tool and a test execution report generator, which are all integrated in one GUI. Other major components are a test model compiler, a test generator engine and a test execution engine (called Spider) [8]. The AML (AGEDIS Modeling Language) test model comprises class, state machine and object diagrams

5 http://www.gentleware.com/

6 http://www.artisansoftwaretools.com/products/artisan-studio/

7 http://www.elvior.ee/messagemagic/general

8 http://www.cs.aau.dk/~behrmann/cora/

9 http://graphml.graphdrawing.org/

10 http://www.omg.org/technology/documents/formal/xmi.htm

11 http://strategoxt.org/Transform/GraphXML

12 http://www.objecteering.com/

13 http://www.altova.com/xml-editor/

(UML 1.4 profile). Classes' behavior are described with state machines. Object diagrams describe initial states of objects and hence of the SUT. AML is annotated with the IF (Intermediate Format) action language to specify methods.

A mapping between model operations and the SUT's interface is provided in an XML file. Test directives provide information about test coverage criteria. Test cases are written in an XML file. AGEDIS supports online and offline testing by providing alternative paths in abstract test case to handle non-deterministic behaviour (useful during offline testing). Its test execution engine (Spider) is able to run test cases on distributed systems in heterogeneous environments. AGEDIS coverage analyzer provides information about possible input values which are not covered and methods which are not invoked by the generated test suite. The defect analyzer provides information about failed test cases and groups them with respect to similar failure reasons.

B.6 ParTeG

ParTeG (Partition Test Generator) is an open source Eclipse plug-in that generates test cases from a test model, created using TopCased[14] Eclipse plug-in, which is composed of a UML 2.0 class diagram and associated state machine diagrams. OCL expressions are used to specify guard conditions. ParTeG constructs a transition tree from the test model (traversing the graph representing the state machine), each path in the tree being a test case. OCL expressions involved in a path are transformed into conditions on input values. (Note that ParTeG does not handle OCL collections or user-defined types.) These conditions are used to define partitions for input values. Values near the boundaries of these partitions are selected as input values for concrete test cases. ParTeG generates test cases in Java which are executed on the SUT using JUnit. Test cases can be debugged using the Eclipse integrated debugger. ParTeG supportes MC/DC, condition, transition, state, multiple condition criteria.

B.7 Qtronic

Qtronic has three main parts: Qtronic Computation Server generates test cases, executes them on model and analyses results; Qtronic Modeler is used to create the test model as UML state machines complemented with an action language, Qtronic Modeling Language (QML), similar to Java and C#. Model can also be created with IBM Rhapsody[15] and Enterprise Architect[16]; Qtronic Client (Eclipse plug-in or standalone desktop application) provides facilities to create test models (using Qtronic Modeler), select test coverage criteria, and analyze model and test suite execution results when they are run on the model. Qtronic provides support for requirement traceability. Test inputs and expected output (oracle) are generated from the test model (state machine and QML): abstract test cases are generated using different algorithms; test cases are generated in C, C++, Java, Perl, Python, TCL, TTCN-3,

[14] http://www.topcased.org/

[15] http://www-01.ibm.com/software/awdtools/rhapsody/

[16] http://www.sparxsystems.com.au/

XML, SOATest[17], Excel, HTML, Word and Shell Scripts. Online testing is done from within the Qtronic Client environment by directly connecting to the SUT using a DLL (dynamic link library) plug-in interface [9].

B.8 Test Designer

Test Designer is part of Smartesting Center solution suite. Smartesting uses third party tools for a number of features: IBM Rational DOORS for requirements definition and traceability, HPQC/HPQP (HP Quality Center[18] / HP QuickTest Professional[19]) to create test adapters, IBM Rational Software Modeler[20] (RSM) for model creation. Smartesting Model Checker and Simulator are integrated with RSM. Other options for model creations are IBM Rational Software Architect[21] (RSA) and Borland Together[22].

A test model is made of a class diagram (to describe data), state machines (state-based behavior) and object diagrams (initial state): theses are all UML 2.x diagrams. The Object Constraint Language (OCL) is used within the class diagram and state machines to formalize transitions between states, guards and the effects of transitions. A test case is composed of a preamble (i.e., a sequence of zero or more operations to reach the behaviour to be tested), a body (the behaviour under test along with the expected results), and an optional postamble (i.e., a sequence of operations to return to the initial state of the SUT).

B.9 Spec Explorer

In Spec Explorer 2010, which we simply refer to as Spec Explorer, the test model is an Abstract State Machine (ASM): a generalization of FSM, written in pseudo-code (i.e., C#-like), to operate over arbitrary data structures. Transitions between states are specified with actions (e.g., functions of the SUT), and rules, which specify what transitions are allowed. In a nutshell, the user defined (abstract) model is transformed into an (internal) model where abstract states become concrete states and actions are given actual input values. This may lead to a state explosion problem: too many concrete states to handle. To reduce the risk, the user can use rules to for instance control input parameter values (e.g., the user can specify a range of integers instead of the whole set of allowed integers), to control the construction of concrete states. Test cases (with oracle checks) can be executed directly on an SUT implemented in .NET. Alternatively, for other SUT implementations a test adapter is required: Spec Explorer

---

17 http://www.parasoft.com/jsp/solutions/soa_solution.jsp;jsessionid=aaa7EAlvipZLSd?itemId=319

18 https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1131_4000_100__

19 https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__

20 http://www-01.ibm.com/software/awdtools/modeler/swmodeler/

21 http://www-01.ibm.com/software/awdtools/architect/swarchitect/

22 http://www.borland.com/us/products/together/index.html

creates the skeleton of the adapter and the user completes its implementation (e.g., calls to the SUT, oracle checks).