Systematic Review of State Based Model Based Testing Tools

By

Muhammad Shafique

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
(OCIECE)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

June 2010

Your file Votre référence
ISBN: 978-0-494-71563-5
Our file Notre référence
ISBN: 978-0-494-71563-5

The undersigned recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis

Systematic Review of State Based Model Based Testing Tools

submitted by

Muhammad Shafique

in partial fulfillment of the requirements for

the degree of **Master of Applied Science**

Professor Y. Labiche (Thesis Supervisor)

Professor H. M. Schwartz (Department Chair)

Carleton University

June 2010

ACKNOWLEDGEMENTS

First and foremost, I offer sincere thanks to my supervisor Dr. Yvan Labiche for all his guidance, patience and knowledge throughout my research. This thesis would not have been completed without his encouragement. I gained a lot of confidence from scheduled as well as casual research and technical discussions with him. His timely feedbacks kept me on right track throughout my research work at SQUALL.

I also thank to my family and friends for their help and encouragement. Their constant moral support and kindness proved a vital catalyst.

Last but not least, I would like to thank all my colleagues at SQUALL for their cooperation and support. I also learnt from them. Thank you all for cheering and caring me.

ABSTRACT

Model-based testing (MBT) is about testing a software system by using a test model of its behaviour. MBT helps in early fault detection and facilitates the automation of many testing activities such as regression testing. To get proper benefits from MBT, (ideally full) automation support is required. This thesis presents a systematic review of prominent MBT tool support where mainly we focus on tools from industry and research which use different state/transition based models. The systematic review protocol describes the scope of this work and the steps involved in tool selection. Evaluation criteria are used to compare selected tool supports 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 systematic review should be of interest to a wide range of stakeholders. First, results should help software companies in selecting the most appropriate model based testing tool for their needs. Second, results should help organizations willing to invest into creating MBT tool support by identifying how to compete with existing support. Third, this study should also be helpful in setting directions of research in MBT since researchers should easily identify gaps between research activities and existing commercial tool support.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

Software testing, that is evaluating software by observing its executions on actual valued inputs [46], 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 [48]. Authors also introduce the notion of grey-box testing that mixes information from the specification and the implementation to derive tests [89].

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 acceptations and they focus on one which is the automation of the design of black-box tests [99]. In this thesis 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: creating test models, 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., means that provide information about the correct, expected behaviour of the SUT [48]). 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 implementation information (from the SUT) 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 [59]—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, is to create simpler, more concrete, more focused (and often more ready to use for testing activities) models [99].

There is more and more interest in MBT since it promises early testing activities, hopefully early fault detection (e.g., [54]). 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 research/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., [58, 99]). 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 5 for further details.)

In this thesis we are interested in helping the above-mentioned stakeholders (i.e., practitioners, tool vendors, and 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 notation 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 thesis we therefore conduct a systematic review of state-based model-based testing tools. This is not strictly speaking a systematic literature review (SLR) [62, 80, 81] 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" [81].

The main contributions of this thesis are:

— We adapted guidelines for systematic literature review (SLR) [81] to our study of state-based MBT tools.

— We provide an extensible set of systematic review criteria to evaluate state-based MBT tools by capturing all important test criteria for state-based models, automation support and test scaffolding criteria. Criteria developed for tool automation and scaffolding support evaluation can be used for all MBT paradigms with no or little modifications.

— We provide a detailed systematic comparisons of state-based model based testing tools.

The rest of this thesis is structured as follows. Chapter 2 discusses the MBT process in general, its benefits, limitations and issues as described by researchers. The reader familiar with MBT may wish to skip Chapter 2. We then present the Systematic Review Protocol we developed for our systematic review (Chapter 3). The Systematic Review Criteria (Chapter 4) describe evaluation parameters using classified attributes. Chapter 5 presents the evaluation and comparison of selected tools. Chapter 6 discusses related efforts to highlight importance of this work. We conclude and describe future work in Chapter 8.

2 MODEL BASED TESTING

Software testing is an activity performed for evaluating software quality, and for improving it, by identifying defects. It is a dynamic verification technique that evaluates a system's behaviour based on the observation of a selected set of controlled executions or test cases. Many approaches to software testing create and execute test cases from source code. We can also apply testing techniques all along the development process, starting from requirements specification and design models [51]. There are many kinds of testing activities, e.g., unit testing, component testing, integration testing, and system testing, that can be classified into structural testing and functional testing. Model based testing (MBT) is about testing from model of functional specifications. Although testing using finite state machine (FSM) models was first introduced in the 50s when literature was first published on this topic [84], MBT got considerable attention from software industry in the mid 90s when companies like IBM and Microsoft started projects to build model based testing tools.

This Chapter first presents the concept of model in MBT (section 2.1). We then detail the steps involved in MBT (section 2.2). Sections 2.3 and 2.4 discuss some of the benefits and limitations of MBT. Issues about MBT identified in the literature are summarized in section 2.5.

2.1 Software Model

The software industry is realizing the significance of models in understanding requirements, building design and guiding other software development activities. The concept of Model is not new in software engineering. In fact models have been built since the inception of computing software [63]. Model Driven Development is believed to be the future approach for creating software systems with less bugs and in less time with low cost.

Models for MBT are abstractions of functional requirements specifications. The ultimate goal of writing a model for MBT is to create tests cases from it. The information

represented by the model is used for test case generation. So it is important for a tester to know software requirements to be able to build model to generate an appropriate set of test cases, i.e., the features that need to be tested need to be modeled. Functional requirements which are not covered by the model are not tested by test cases generated from the model. A test model can come from some kind of design documentation of the SUT: the test model is a design model. Alternatively, a test model can be simpler, sometimes more abstract, than a model completely describing the System Under Test (SUT) to (1) focus on specific SUT functionalities, and (2) manage complexity. In this thesis, we assume test models are simplifications of the functional requirements.

2.2 Model-Based Testing Process

MBT, which is a black box testing approach, is used to test functional aspects of software systems using models. It is a complete process which involves several activities. In this section, we walk through these activities in detail. It is important to note that MBT is done using some automated tool support, which means that all (or most of) the following activities are automated. However, different tools provide different level of automation. A tester could use more than one tool to automate the whole of MBT process.

2.2.1 Build the mental model

MBT is a new activity even for experienced testers. It has introduced a new and important additional responsibility for a tester which is building model specifically to test functional requirements of a SUT. Software organizations report that this is a difficult task for experienced testers more so than for younger testers [19, 28]. A model-based tester needs to have some kind of programming skills. Creating a test model is just like writing a program in some high level programming language.

El-Far in [63] defines a few activities, prior to MBT, which he collectively calls "building the mental model". According to him, it is important to understand system requirements before starting building the model. It is helpful to focus on important aspects of the system which a tester might want to test using MBT. El-Far provides the following tips to testers on the kind of activities that may be performed to improve system understanding:

- **Start exploring target areas in the system to develop technical in depth understanding:** This could be done by talking to teams which are assigned dedicated tasks. System understanding is necessary, especially in test case fault analysis.

- **Gather relevant useful documentation:** Model-based testers should learn about the SUT as much as possible through requirements specifications, design documents, technical manuals and user documentation. Some of these documents might not be available or might go through massive changes during multiple iterations.

- **Establish communication with requirements, design and development teams if possible:** This is helpful in getting answers of questions which a tester might have about the SUT. It also saves a lot of time because many ambiguities in natural language and formal documents can be better resolved by direct contact with the relevant person.

- **Study the domain of each input:** Understanding of input values is important to generate test cases. Indeed, it is important for a tester to understand legal, illegal, boundary and normal values of an input parameter before building the model.

- **Document the conditions under which a response occurs:** Response of a system is an output to the caller (test adapter). A tester should study and document all conditions which trigger different actions.

- **Study inputs of the sequence that need to be modeled:** A tester should understand scenarios that the SUT accepts and does not accept.

- **Model independence:** When creating the test model, a tester should not be biased or get influenced by the development model (if any). Test model should be created separately, though from the same set of requirements. On the other hand, a tester should constantly communicate with the development team to get information about any ongoing change in system requirements.

2.2.2 Build the model

After doing the mental model, the next most important and creative activity during the MBT process is writing the model using a specific notation. Of course, a tester needs to understand requirements specification thoroughly to be able to create a model for test cases generation. Experiences have shown that the model building activity is helpful in identifying errors in requirement specifications. The process of understanding requirements specifications written by different teams or individuals with different mindset helps to highlight problems even before starting the testing process [27].

When building the model, different abstraction mechanisms can be used:

- **Function abstraction:** The model omits some less important and simple functionalities of the SUT. Utmost care should be taken while deciding which function is simple and less important. And this should not be based (solely) on volume of code a function body might have. A one line functions could trigger a ripple of calls to some important functions of the SUT.

- **Data abstraction:** This means abstractions of input and output. Input abstractions means that the model omits or simplifies some inputs of an SUT operation. Output abstractions means the model simplifies or omits the outputs of the SUT operations.

After carefully deciding (and documenting) abstractions, the next step is selecting a suitable modeling notation to create the model. This decision is influenced by available MBT tools and the notation(s) they support [99]. We provide an overview of different modeling notations available to a tester in Section 4.1.

After building the model, it is important to verify the model is correct. Many tools are available which provide model verification capabilities.

2.2.3 Generate abstract test cases

After building the test model, the next step is test case generation. These test cases are called abstract test cases because they are generated form the test model, which is created using abstractions: e.g., a test adapter will be needed to eventually execute the corresponding test cases. The most important concern in this step is high coverage of the model with a minimum number of test cases. Depending on the coverage criterion (see details in Section 4.2), the number of test cases can be very large or even infinite. For economic reasons, it should be reduced to a minimum, and for quality reasons it must be sufficiently high to reduce the number of undetected failures [91]. Before generating test cases, a tester has to select a coverage criterion (or criteria)

Some tools generate input and expected output values for abstract test cases. The output value is used as oracle. Some MBT tools provide traceability between functional requirements and generated abstract test cases [4]. This indicates how well a generated test suite covers functional requirements. If a test case fails, requirements traceability can be used to trace to the related requirement, to investigate whether the problem is with the requirement, test model, or the SUT.

2.2.4 Execute test cases

Abstract test cases cannot be directly executed on the SUT because of the abstraction of the test model. To adapt the abstract test cases to the system, it is necessary to insert needed information into the abstract test case by using a mechanism referred to as test adapter [92]. A test adapter transforms the structure and data of an abstract test case into a concrete test case that can be executed on the SUT. Conversely, the adapter must collect concrete SUT outputs and abstract them to compare with the expected output.

A test adapter makes abstract test cases independent of the implementation, which means that by simply changing the adapter details, an abstract test case can be executed on any implementations.

2.2.5 Analyse test results

The last step in the MBT process is the analysis of test results. A test case might pass or fail. A tester must analyze the reasons for failure(s). The test engineer should not forget that a failing test case may be the result of a fault in the model, in the requirements, in the test adapter, in the oracle, as well as in the SUT.

Utting and Legeard in [99] outline their observations about test cases failures. A first analysis of failing test cases shows that the main cause is the test adapter code. Half of the remaining test case failures are due to the SUT, while the rest are due to errors in the model or requirements. This ratio can change widely depending upon testers' experience, domain of the software, level of testing automation, rate of changes in the SUT and requirements.

2.3 Benefits of Model-Based Testing

MBT has been around for more than three decades. However, it has only got some acceptance in industry in the last fifteen years when major players in the software industry such as IBM and Microsoft started MBT tool development projects after realizing the effectiveness and benefits of MBT [54]. Researchers and practitioners have outlined numerous benefits of MBT. In this section, we describe some of these benefits, as outlined in [63]:

- **Communication among teams:** Modeling is a precise communication tool among teams. Certain models are sometimes excellent vehicles of presentation to non-technical management. The model is a living accurate document of the aspects of the system it expresses.

- **Intensive automated testing:** By using an MBT tool, one can easily generate a large number of test cases. Tests could be based on what a testing team would like to test based on system requirements and test model. Tests could also be based on what an end user would like to do with the system function.

- **Failure exposed by MBT:** Test cases are generated from models using test criteria. A good MBT tool generates test cases using more than one coverage criteria. This causes different combinations of input values to the SUT and hence more chances to find faults.

- **Model verification:** Most MBT tools provide facilities to verify a model using some formal verification method. A tester does not need to learn how to formally verify a model because the MBT tool hides that complexity. This gives confidence to the tester about the correctness of the developed model.

Utting and Legeard elaborate the following benefits in [99].

- **The SUT fault detection:** MBT detects more faults than manual testing. This is also evident from statistics available from companies like Microsoft [60, 97], IBM [69] and Sun Microsystems [47].

- **Reduced test cost and time:** MBT requires less time and cost than manual testing. This claim is supported by several case studies. The costs of using MBT to test functions of a digital switching system was 90% less than the manual testing. In the smart card industry, a study shows a reduction of 25% in testing cost when compared with manual testing. Because it is automated and systematic, MBT also requires less time in analysing the cause of a failure [99].

- **Improved test quality:** The quality of manual test cases is highly dependent upon the expertise of test designer. It is difficult to be certain about test quality when requirements change rapidly in an iterative agile development environment. This difficulty can be diminished by using MBT with some suitable tool support. Automated test cases generation without human intervention improves test quality, saves time and gives confidence to testers and development teams.

- **Requirements defect detection:** Requirements are written in natural languages. It is highly likely that the document might contain some contradictions and unnecessary overlaps. Some important information might be missing or written in

a wrong way. Since test models are created from requirements, it is important for a tester to have a clear understanding of functional requirements. Experiences have shown that many faults are detected in requirements specification during the test model building phase.

- **Traceability:** Since a test tool automatically generates tests, it can keep track of traceability links between requirements and test cases as well as with test execution results. This information can be used to identify paths which are not covered by generated test cases, which requirements are not covered, and requirements that are not properly met.

Robinson in [93] briefly describes the following benefits.

- **Developers' satisfaction:** Developers do not care about test case creation and maintenance. They are pleased to see that their code is well tested. Since MBT detects errors even at an early stage, developers get early feedback about their code which helps them streamlining their work.

- **Management satisfaction:** Management could see that testers are creating more test cases using models. This increases their level of confidence on development and testing teams.

Boberg in [54] reports early fault detection in E-mail gateway and Instant Message gateway subsystems using MBT. Hartig et al. in [70] report their ideas of using MBT in regression testing. They suggest a Model-in-the-Loop (MiL) test environment for this purpose. Paradkar in [88] describes effectiveness at detecting faults of MBT test generation techniques. He used BB-TT formal method [88] and mutation based techniques to generate test sequences. He argues that this approach proved more effective than those based on boundary values and predicate coverage criteria for transitions. Gravell et al. in [67] describe their experiences related to concurrent modeling and development in different case studies. They emphasize that testing is important and MBT is an excellent choice. Braspenning et al. in [57] describe their experiences using methods and techniques to reduce integration and testing of software components using model-

based integration and testing. They used models of different components to perform verification, validation and testing at an early stage. Their approach enabled early detection and prevention of problems that would otherwise have occured during real integration.

2.4 Limitations of Model-Based Testing

Dijkstra famously remarked that "testing can be used to show the presence of bugs but never to show their absence" and this also applies to MBT. Utting and Legread in [99] outline the following limitations of MBT:

- **All fault detection:** MBT cannot guarantee to find all the faults.
- **More tools:** MBT is a comparatively new and sophisticated testing approach which requires a good automation support with help of a testing tool. Although some tools are available, there is still a need to improve the level of automation.
- **Outdated requirements:** In an iterative development process and evolving software, requirements soon become outdated. If a model is built using outdated requirements then the tester will find many errors in the SUT.
- **Time to analyze faults:** When a test case fails, the tester has to do a thorough analysis to find the root cause of the failure. The error might be in code of the SUT, in the test adapter, or in the model or requirements. This analysis process may require a lot of time because the tester has to explore requirements, test adapter code and the model to pinpoint likely reasons for faults.
- **Useless metrics:** When test cases are designed manually, the number of test cases is considered a measure of how testing is progressing. With MBT, it is easy to generate hundreds of test cases using a tool which makes test metrics useless.

2.5 Issues and Future Challenges

MBT needs proper planning, training and time investment. To get real benefits from MBT, a software organization needs to have experienced model based testers. Otherwise, there is a need to invest time and money on training. A software organization might have a well defined customized process model. Introducing MBT as a new activity might not be an easy task for many reasons.

In this section, we describe major obstacles and drawbacks with MBT, as outlined in [63, 64]:

- **Lack of expertise:** Model based testers need a proper understanding of models and underlying theories and mathematical concepts. A tester needs to have good level of expertise in tools, scripts and programming languages.

- **Management support:** A solid planning, acknowledged and enforced by management, is required to handle rapidly changing functionalities, frequent iterations, and frequent rebuilding of models.

- **Tight deadlines:** MBT is a reliable yet time consuming activity. It seldom (in fact never) happens that a system has very few bugs at the beginning. In MBT, bug tracking is a relatively time consuming activity as compared to other testing approaches. Enough time must be allocated to MBT.

- The Organizations that want to test using a model that has no tool support should consider developing their own tools. Frameworks are available which can be used to developed a tool according to needs [5, 8, 13, 21, 22].

- **State Space Explosion:** In a large complex system, it is quite possible that the number of states, rules and variables is also large. This consequently generates millions of combinations and hence of test cases, which cause state explosion. Several techniques have been proposed which do not always work [95].

Robinson in [93] outlines the following obstacles to MBT:

- **Testers are often non-technical:** Testers in many companies are non-technical people. Their limited technical background constrains the amount of test innovation (e.g., MBT) that can be absorbed by the team.

- **Lack of latest knowledge:** Testers do not necessarily read latest texts, especially research papers, published to introduce new trends, approaches and techniques. This keeps them away from latest advancements in the field. This is also true for software companies which develop MBT tools. There is a gap between what is available in the area of MBT research and what is provided by commercial MBT tools.

Heimdahl in [74] describes experiences and challenges with MBT:

- **Model Coverage criteria:** MBT tools generate test cases from a test model using test coverage criteria. A tester can get hundreds and even thousands of test cases instantaneously. A test suite with many test cases should not mean better test adequacy though. It has been experienced that sometimes a test suite does not cover some important parts of the SUT even when using several criteria together. This shows that there is room for improvements in the area of test model coverage criteria.

- **Problems in model and tool selection:** Available MBT tools use different test model notations and have their strengths and weaknesses. Choosing the right MBT tool is therefore not an easy task, even though some (incomplete) systematic reviews and studies [85-87, 100] exist.

- **Distributed systems:** MBT of Distributed Systems is an area of current and future research which is rich of challenges and possible accomplishments because of the specifics of these kinds of systems, such as their specific architectures (e.g., they are often embedded), the communication protocols they use, synchronization issues, and the importance of non-functional aspects (e.g., security) [95].

- **Non-functional requirements:** Traditionally, a model in MBT models functionalities, and therefore cannot be used to test non-functional requirements such as verifying there is no deadlocks or race conditions, testing for performance, testing for fault-tolerance. Denaro et al. in [61] list some parameters which a test case should have to perform model based performance testing of a distributed system.

3 SYSTEMATIC REVIEW PROTOCOL

A systematic review identifies, evaluates and interprets available material to answer a research question [81] by following a review protocol which is developed before the start of the review process. Adopting guidelines for systematic literature reviews in software engineering [81], our review protocol precisely describes the scope and objectives of the review (section 3.1), the research question(s) (section 3.1), the search process (section 3.2), inclusion/exclusion criteria of identified tools (section 3.3) and the evaluation criteria (section 3.4). Our study is mainly about MBT tool support where we focus on tools and use research literature as support material to know MBT process and expected automation level. For this reason, we studied considerable research and technical material to collect parameters (criteria) to conduct primary study. We use these parameters to conduct our systematic review by using Systematic Review Criteria (chapter 4) which is part of Systematic Review Protocol. A summary of the search is provided in section 3.5.

3.1 Scope, Objectives, and Research Questions

The scope of our systematic review is state/transition based MBT tool support available in research and commercial domains. Note that we use the term MBT tool support since some of the identified tools are actually frameworks. However, unless otherwise specified, we will use the term MBT tool in the rest of this paper.

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 MBT tool. Last, results should help researchers identify possible gaps between current MBT implementation and MBT research.

To achieve these objectives, the research question we are interested in is:

- What are the capabilities of state/transition based MBT tools in terms (for instance) of test coverage, automation level and test scaffolding construction?

Kitchenham [81] recommends to break down research question into facets to ease the systematic review. We, therefore, divide our research question into: (1) MBT tools search, and (2) evaluation to analyze their capabilities.

3.2 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 [81]) and Google search engines. 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 [76, 82, 83, 99], and 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 web sites of projects and private organization (e.g. AGEDIS [1], D-MINT [6]). The members of the model-based-testing mailing list are parishioners and researchers who actively share information about existing and new MBT tools, MBT adoption and latest research in the area. 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 (non-) academic, non-IT venues, specifically in Business Source Complete (scholarly business database). The search ended in February 2010.

Our search process is summarized below:

1. Identify search strings to perform electronic searches: see section 3.4.
2. Study selected paper abstracts and introductory sections to identify relevance with our work.
3. Scan list of paper references, authors and/or research groups and projects web sites to collect more information about MBT literature and tools (if any).

4. Collect tool user manuals, white papers, technical reports, books, case studies, video tutorials and demonstrations, presentation, product descriptions and customer feedbacks available on the vendor website and the Internet.
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.

3.3 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 study in [81]. To select primary studies accurately, inclusion/exclusion criteria are set with respect to research questions. We define 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/transition 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) and Simulink/Stateflow chart
3. All tools which automatically generate test cases from test models.

3.4 Search Strings

The iterative search strategy to collect tools for primary study and support material is described in section 3.2. 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 questions.

Kitchenham [81] identifies six groups of terms to break down research question into individual facets to facilitate the search process. They are population, intervention, comparison, outcomes, context and study design. Based on the research question described in section 3.1, 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 Terms2. In the first phase of our search process, we constructed sophisticated meaningful strings (see Appendix B) from Intervention Terms1 with Boolean AND and OR operators.

After considerable search iterations with resources mentioned in section 3.2, strings (Intervention Terms2) were identified. In the second phase of our search process, we constructed meaningful search strings from Intervention Terms1 and Intervention Terms2 (see Appendix B) to collect tool specific material in form of papers, technical reports, tool manuals, case studies, presentations, online demos, video lectures and online discussions.

Strings in Intervention Terms1 were general MBT-related terms. The first phrase of the search led to the identification of MBT tools. The tool names were then part of the Intervention Terms2 so as to refine the search.

3.4.1 Population Terms

State/transition model based testing tools.

3.4.2 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, SDL, Specifications Description Language, using, of, for and with.

3.4.3 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, TGV, Time Partition Test, TorX, expecco, PrUDE, KeY, JUMBL, UniTesk, Escalator, MOTES, AGEDIS, GOTCHA-TCBeans, Uppaal TRON, TestEra, exprecco, AsmL, IBM, Rhapsody, Tau, Statemate, SDL, Testing and Test Control Notation (TTCN-3[1]), suite, Automatic Test Generator, ATG and TestConductor.

3.5 Statistics gained from Search

In total 30 tools and 2 APIs were found after completing the search process in February 2010. After applying inclusion/exclusion criteria, we selected 15 tools and one API for evaluation and finally 12 tools as primary study. The list of tools selected for primary study is: GOTCHA-TCBeans [12], mbt [18], MOTES [7], NModel [22], Spec Explorer 2010 [27], TestOptimal [32], AGEDIS [1], ParTeG [23], Qtronic [4], Test Designer [29], Reactis [25], Simulink Tester for T-VEC [36], ModelJUnit (API) [21], IBM Rational Automatic Test Generator (ATG) [42], IBM Rational SDL-TTCN Suite [43] and Test Suite Generation/Reduction (TSGR) [30].

Three tools (Reactis, Simulink Tester for T-VEC, and TSGR) and one API (ModelJUnit) are not included in results comparisons (section 5.2) because of reasons outlined below.

Reactis and Simulink Tester for T-VEC are not included in results comparisons because of the hybrid nature of the test model, which consists of Simulink blocks and Stateflow charts. Simulink is considered a data flow notation while Stateflow chart is a state/transition based modeling notation. Additionally, these notations are used to model mostly continuous phenomenon as opposed to state-based behaviour in software. This systematic review is mainly about state/transition based test models and we believe comparison of Simulink/Stateflow chart with other notations like (E)FSM, ASM, UML state machines would not be a fair comparison. We believe that simulink-based tools

[1] http://www.ttcn-3.org/

belong to a specific family of MBT tools and should be compared with one another, likely with dedicated comparison criteria.

TSGR is a research tool that generates test cases from EFSM to cover requirements specified in the model. It can also reduce the size of the test suite by eliminating duplicate test cases (transition sequences). The tool generates sequences of transition labels to cover requirements, that is abstract test cases without information about the actual test method names, inputs, and expected output values (oracle). TSGR does not execute test cases on the SUT neither does it provide information about third party test execution tools [30]. ParTeG is another research tool (see section 5.1.8) which uses third party tool to create models, generates test cases, uses third party tool (JUnit) to execute tests and results analysis. It uses the Eclipse debugger for test case debugging. We selected research tools like ParTeG which are ready to use. Although TSGR generates abstract test cases, it does not provide sufficient automation support to be selected in our study.

ModelJUnit is a Java based API that allows online test coverage of states and transitions of (E)FSM models. It is the responsibility of the tester to provide implementation of test generation algorithms. There is no support for automation and test scaffolding creation [21]. Given the little support for MBT, we decided to not include ModelJUnit in the comparison.

The list of tools and APIs which are excluded from primary study is: TDE/UML [26], MaTelo [16], ATD-Automated Test Designer [3], errfix [8], GATel [11], TGV [33], Time Partition Test [34], TorX [35], expecco [10], PrUDE [24], KeY [15], JUMBL [14], UniTesk [5], Escalator [9], Uppaal-TRON [37] and TestEra [31].

Although this systematic review is about MBT tools and APIs, we studied more than 328 research papers, technical articles, online manuals, books and presentations and selected 78 of them as supporting material to better understand the domain.

4 SYSTEMATIC REVIEW CRITERIA

To extract information from primary study to answer research questions, a data extraction procedure as precise and accurate as possible (to avoid bias) needs to be defined [81]. 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 4.2), *Tool Automation Criteria* (section 4.3), *Test Scaffolding Criteria* (section 4.4) and *Miscellaneous Criteria* (section 4.5). An initial sub-set of those criteria had been established prior to conducting the evaluation of selected tools (primary study), and was then refined given what we learnt about the tools: the identification of criteria was iterative.

Note that gathering information about primary study was not always an easy task. There are several reasons for that. Research 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 important information for the 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.

4.1 Types of Notations

MBT requires model to generate test cases. There are hundreds of different notations available to build the model. Utting and Legeard group available prominent notations in six paradigms [99]. We briefly describe all these paradigms here.

State or Transition-based Notations: These notations describe a system using different states of the system. States are connected using transitions. This paradigm includes FSM, EFSM, Harel statecharts, the UML State machines, LTS, Simulink/Stateflow charts, ASM and Specifications Description Language (SDL) models. In our current systematic review, we are focusing only on those MBT tools which use state or transition-based notations.

Pre-Post Notations: These models are collection of variables to represent internal state of the systems with operations which modify them. Z, B, VDM, Alloy and JML are examples of these notations.

Functional Notations: These notations describe a system using mathematical functions. The functions may be of first order only.

Operational Notations: These notations describe a system as a collection of parallel executing processes. Petri Nets is one example.

Stochastic Notations: These notations describe a system by using probabilistic model of events and inputs and used to model environments than the SUT. Markov Chains is an example of these notations.

Data-flow Notations: These notations are used to represent data rather than control flow. Examples are Lustre and Matlab Simulink.

4.2 Test Criteria

This section provides information about test coverage criteria that can be selected to generate test suite from state/transition based model (reader is referred to Appendix A for introductory information on state/transition based models). All these test criteria are applicable to state/transition based models and none of them is related to code coverage of the SUT. It means a tester could use any or all of them to generate test cases for black box testing. There is some overlap in the names of MBT and white box testing coverage criteria. However, their semantics are different. For example, in white box testing, statement coverage criterion is used to measure how well statements written in code are

covered. In MBT context, this coverage criterion is used to measure the coverage of statements written in the model [99].

Test Criteria are further classified into: Adequacy criteria (AC) and Coverage Criteria (CC). An AC criterion is used for test case construction while a CC criterion is used to measure an existing test suite. There is a wide variety of terms used in primary study to refer to (sometimes the same) criteria. We tried to recognize similarities, synonyms (similarly to [46]) and used MBT tools terminology for uncommon criteria. Depending on tool support, some criteria could fall in AC as well as in CC.

We divide Test Criteria into three groups, i.e., Model Flow, Script Flow and Data and Requirements coverage: see below.

For each group of criteria, or 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.

4.2.1 Model Flow Coverage Criteria

Many coverage criteria have been developed for state/transition based test models. We select State, Transition, Transition-pair, Implicit Consumption or Sneak Path, All Paths, Parallel-Transition, and Scenario Criteria for this systematic review. All these are adequacy criteria, i.e., they are used to build test cases. The first five are well-known criteria [46, 52] while the other two are not commonly used but are effective to cover important aspects in MBT tool support. A good MBT tool supports more than one coverage criterion.

States coverage: According to this coverage criterion, all states of a transition based testing model should be visited at least once. A state could be visited more than once by different (or the same) test case but too many such visits may indicate that test generation algorithms is inefficient. States coverage is 100% if all state are visited.

Transition coverage: According to this coverage criterion, all transitions of a transition based testing model should be visited at least once. A transition could be visited more than once by different (or same) test case but too many such visits indicate that test generation algorithms is inefficient. Transition coverage is 100% if all transitions are visited.

Transition pair coverage: Transition pair or 2-Transition coverage criterion measures the coverage of each two adjacent transitions at least once.

Implicit Consumption/Sneak path: This coverage criterion (in UML state machines and EFSM) allows receiving messages that are not handled by the current state and discarded automatically [39].

All Paths: All paths requires the coverage of every possible sequence of state, transition and/or control flow branches at least once. This coverage criteria generates a huge number of test cases.

Parallel Transition coverage: According to this coverage criterion, each parallel transition configuration in model should be covered at least once. Parallel transition coverage is helpful to test systems which have concurrent behaviours. Such behaviours are complex to design in FSM/EFSM.

Scenario coverage: A tester defines test scenario by selecting states and transitions (with selection order) to create scenario (test case) [53]. Scenario coverage criterion measures the coverage of all created scenarios at least once.

4.2.2 Script Flow Coverage Criteria

These criteria refer to Interface (function), Statement, 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 scripting languages; 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

[46, 99] 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.

Interface function coverage: This criterion refers to the coverage of those methods which are called on the SUT or present in a test model class [72].

Statement coverage: Every statement in model and/or in a script file should be executed at least once. A test case covers a statement if it executes the statement. This coverage criterion could be considered as white box test coverage of the test model and attached script.

Decision (or Branch) coverage: This coverage criterion measures the coverage of each decision (also called control point) with true and false values. A decision point is analogous to an if-else block. The entire Boolean expression is considered as a whole, regardless of whether it contains logical operators such as AND, OR. A test suite should test both if and else blocks (one when conditional expression is true and one when it is false). In other words, there are two test objectives for each decision/if-then-else.

Condition coverage: A condition is a Boolean expression which can not be broken into simpler Boolean expressions. If a decision is composed of several conditions (with AND, OR, …), Decision (or Branch) coverage cannot be sufficient. Condition coverage is about the true and false outcome of each condition. A test suite should achieve true and false results for each condition. Note that achieving condition coverage does not necessarily imply decision coverage.

Modified condition/decision coverage: This coverage criterion requires that the outcome of a condition independently effects outcome of a decision. A condition independently effects decision if varying that condition changes the result of the whole decision while keeping all other conditions unchanged. For example, for a decision expression like (condtion1 && condition2 || condition3), a maximum of six test cases are required (2 for each condition).

Atomic Condition coverage: According to this coverage criterion, a test suite should cover all possible combinations of condition outcomes of each decision. This requires up to 2^n test cases where n is the number of conditions in a decision. This test criterion generates a huge number of test cases and is also referred to as Multiple Condition Coverage.

4.2.3 Data and Requirements Coverage Criteria

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. Requirement criterion is used to cover requirements by generated tests [99].

One value coverage: This criterion requires to test one value from domain of input test case parameter. Testing one input value might seem infeasible but in combination to other input parameters, good coverage could be achieved.

All values coverage: This criterion requires to test all possible domain values of an input parameter. This test criterion is rarely used.

Boundary value coverage: This test criterion requires from a test suite to cover every arithmetic comparison at least once. This means that for comparison like (==) or (!=), three test case should be generated. One for equality and two for inequalities. This also means that for comparison like (<, <=, >, >=) four test cases should be generated. Two test cases for values within boundary and two test cases for values outside it [39].

Pair wise coverage: This criterion requires a test case for each possible combination of values of all pairs of input parameters to a method.

Requirements coverage: Selected functional requirements are incorporated in test model using requirements IDs. It is highly desired to ensure that all those requirements which are linked to the model must be covered otherwise generated test suite is not adequate

and some parts of the SUT might remain untested. Requirements criterion requires that all requirements linked to the model are covered.

4.3 Tool Automation Criteria

Tool Automation Criteria measure the automation level of a model based testing tool. These criteria provide valuable information in tool selection as high level of tool automation is one of the most important requirements of a software organization. We have divided these criteria into seven criteria and there salient details are provided in this section.

These criteria 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.

4.3.1 Model Creation

This criterion measures the visual modeling facility provided by a test tool. GUI support in test model creation is extremely helpful in creating large and complex models. Test model is saved in some readable and processable textual format (mostly in XML). If a

model does not provide visual modeling facility, then the tester has to create the model himself, which requires good skills and knowledge of the underlying language. Furthermore, it is highly likely that tester will do mistakes in creating large models.

4.3.2 Model Verification

Test cases are generated from test model and test model is created from functional specifications. Utting and Legeard in [99] report their experiences about test cases failure. According to them, one of the reasons of test cases failure is the faulty model. An MBT tool should provide model verification facility to help tester verify the model before generating test cases.

4.3.3 Sub-models

Large and complex systems are hard to test because of a huge number of test cases and a high risk of not testing some important functionality. This problem could be solved by dividing test model into sub-models before generating test cases. The tester can generate separate test suites for each sub-model (component or subsystem) or can combine them before generating test cases. However, in first approach, more than one test adapters are needed. A tool should provide automation support to create and then integrate sub-models before the start of test generation process.

4.3.4 Test Case Generation

Model based testing is black box testing and test cases are generated from test models. The size of the test suite in model based testing is much bigger than other manual and automated testing techniques. It is hard and sometime impractical to generate test cases manually in MBT. A tool should provide facility to generate test cases from test model using selected coverage criteria.

4.3.5 Test Case Execution

The generated test cases are called abstract test cases because they are generated from system abstractions (test model). It is hard to run abstract test cases manually because the

tester needs to know everything about the SUT to concretize test case. There are separate tools available to run abstract test cases on the SUT which means a tester has to learn those tools as well. We emphasize that a MBT tool should generate and execute test cases. This coverage criterion measures this capability.

4.3.6 Test Case Debugging

A test case failure analysis is a tedious work in model based testing. This problem is hectic when the test model is large and complex. In high level programming languages, a good development environment provides a program called debugger to track runtime errors. Debugger makes a programmer's life easy by providing different facilities. A programmer can execute program step by step using break points and declared variables could be observed using watch window facility. Similarly, a model debugger should provide tester facility to debug abstract test cases to know the failure reasons. With test case debugger, a programmer could easily detect failure reasons by using break points and step by step test case execution.

4.3.7 Requirements Traceability

A Requirements traceability matrix maps functional requirements with generated test cases. It provides three main benefits to model based testers. First, a tester could see which requirements are still to be tested. Second, requirements of a failed test case could easily be tracked. Third, mapping relationship among test cases and functional requirements could also be found. It is highly desirable that a test tool should provide end-to-end requirements traceability.

4.3.8 Regression Testing

Regression testing is the process of testing changes to software to make sure that unchanged (along with changed) parts of the software work. Regression testing in MBT is needed when requirements or model are changed. This requires re-execution of test cases, regeneration of test cases and/or modifications in test adapter (driver) code.

If M' is a modified version of model M when some changes/modifications are made to functional requirements, the corresponding test suites are T' and T respectively. T_r (the reusable tests) is a subset of T that could be rerun to test parts of the model M that remain unchanged in M'. Then $T' = T_r \cup T_n$, where T_n are the test cases generated from the modified/new parts of the test model. T_n is also called the set of retestable test cases. Some test cases valid for M might be invalid for M' because of the modifications made to M. These test cases are called Obsolete test cases (T_o). $T' = T_r \cup T_n \cup T_o$.

A regression testing criterion in the MBT tool context measures the ability of a tool to generate test cases to perform regression testing without tester's intervention. The tool should indicate retestable, reusable and obsolete test cases.

4.4 Test Scaffolding Criteria

Test scaffolding criteria measure adapter and stub creation, oracle automation, offline testing and online testing. A tool can provide full (F) or partial (P) or no (N) support for these criteria. Support can also come from a third party tool (TP). 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.

4.4.1 Test Adapter Creation

A test adapter or test driver is a small self contained piece of code which is responsible to execute test cases on the SUT. An abstract test case includes sequence of input parameters and calls to methods of the SUT. Input parameters are in abstract form because they are generated from higher level of abstractions (i.e. model). It is responsibility of test adapter to provide missing information to parameters before executing test cases. This process is called concretization. Similarly, test adapter extract abstract information from return values before a verdict is made by the oracle. Test adapter automation is an important requirement of a model based tester. Almost all of

MBT tools need some input parameters before generating test adapter code. Test Adapter criterion measures the ability of a tool to execute abstract test cases on the SUT without adding any code manually.

4.4.2 Test Oracle Automation

The oracle provides the ability to automatically determine whether a test case is passed or failed. Oracle information should be part of test models. Information about oracle is not a mandatory part of functional requirements although understanding of functional requirements is used to add oracle in a test model to make verdict about a test case [63]. If a tool does not provide test oracle automation, the tester has to provide this information by manually editing the abstract test case. Test oracle automation measures two things: first, a test tool should generate expected output from the model; second, it should make verdict without the tester's intervention.

4.4.3 Stub creation

In model based testing, it is occasionally required to emulate unavailable components to execute tests on the SUT. Stubs of unavailable components can be added in test adapter to use them during test execution phase. Different patterns are available for this purpose [94]. A tool should provide stubs generation support.

4.4.4 Offline Testing

In Offline testing, test cases are generated before they are run. One obvious advantage of offline test case generation is multiple execution of same test suite on different versions of the SUT. A tester does not need to generate new test cases as long as there is no change in test model or test coverage criteria. A repository of offline test cases can also be maintained on server which can be used by different testers without knowing test model complexities.

4.4.5 Online Testing

In Online testing, generation and execution of test cases conducted at the same time. Test case generation algorithms react on the output of the SUT. It is extremely helpful in testing non-deterministic behaviour of the SUT. Test model and the SUT both can have non-deterministic behaviours. In online testing, a test generator could see which path the SUT has taken after the point of non-determinism. It then follows the same path in test model to generate test case. If a test tool does not do online testing then it is not possible to test non-deterministic behaviour of the SUT.

4.5 Miscellaneous Criteria

Miscellaneous criteria consist of soft evaluation 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 focus on technical aspects rather than process or business/financial ones: 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).

4.5.1 Modeling Notations

As described previously, this systematic review is about state based MBT tools. Different state based notations have their special characterises. For example, the UML state machine can be used to model any system, LTS is considered better to model reactive systems, FSM is used to test only flow aspects of a system. Furthermore, a software company would also like to know modeling notations supported by an MBT tools to access the tool usability for domain specific software projects.

4.5.2 Tool Category

An MBT tool may be commercial developed by a software company or it can be an open source tool developed by some community. A research tool may be developed by some

consortium or university. We also consider academic prototype tool, which are available for evaluation, as research tools. Software companies usually do not share technical information but a research group usually like to collaborate with other research groups or individuals to improve research area and tool. Commercial tool also means more reliability, functionality and automation with high licensing cost. So tool category information could be interesting for many readers.

4.5.3 Software Domain

A MBT tool could be domain specific or for general use. There are tools good for MBT of real-time system, interactive systems and automobiles systems. A domain specific tool may provide more domain specific test coverage criteria than a general tool. There may be some domain specific modeling notations as well to generate better test cases. On the other hand, those companies which develop software for more than one domain may want a single MBT tool to cover "All" domains. Information about MBT tool domain is also important in selecting a tool.

4.5.4 Target Platform

Target Platform defines the programming language(s) used to develop the SUT. A test case execution support (test adapter or external engine) may execute test cases on specific or all programming languages.

5 EVALUATION

In this chapter we describe our findings about state based MBT tools. We used the evaluation criteria that we discussed in section 4, including *Test Criteria*, *Tool Automation Criteria*, *Test Scaffolding Criteria*, and *Miscellaneous Criteria*.

This Chapter is structured as follows. First, we succinctly discuss the tools we have identified from our selection procedure, commenting on their main capabilities (Section 5.1). Then we compare these tools using the evaluation criteria in sections 4.2 to 4.5, one set of criteria at a time.

5.1 Tools

This section provides information about selected state based MBT tools. Each sub-section is dedicated to one tool. All these tools are either research or commercial or open source and comprehensive information about each tool is needed to be able to understand their evaluation and comparison. We studied technical articles, white papers, case study reports, online videos and dedicated online users forums to get important information for evaluation process.

5.1.1 GOTCHA-TCBeans

GOTCHA-TCBeans (Generation of Test Cases for Hardware Architecture - Test Case JavaBeans) is a model based testing tool from IBM. GOTCHA part of GOTCHA-TCBeans generates abstract test cases from the test model using a set of testing directives while TCBeans is a Java based framework that supports test case execution on the SUT. GOTCHA-TCBeans is developed for internal users and is not available for commercial or academic use. This tool runs on Windows, IBM AIX and Linux operating systems.

Test model in GOTCHA-TCBeans is FSM which is generated from machine specifications (model program) written in GDL (GOTCHA Definition Language). GDL

is an extension to MDL (Murphi description language[2]). Machine specifications in GDL comprise of three parts with additional test constraints and special marks. Test constraints define forbidden states, paths/transitions in test model to avoid state space explosion. Special marks are used to provide additional information like starting and ending test states of the model.

The first section of the specifications describes declarations of constants, types, and state (global) variables. The purpose of constants is more or less same as constants in high level programming languages. That is, they are used for comparison with some value or used to define size of some data structure e.g. an array. A tester can define a new data type (e.g. class, enumeration) using Type construct. A Type data type in GDL is similar to an instance in Java class. State variables are used to store state information of the FSM. The second section contains functions which are used/called on the SUT. The third section contains valid method calls (stimuli) specifications and the responses to these stimuli (oracle). These are called transition rules. Each transition rule has an associated action (method to be tested on the SUT). An action can have parameters, conditional expression and loops. Each machine specification normally has more than one rule which is called rulesets. GOTACHA generates FSM from machine specifications with all possible input values. This process is called "state exploration" and the possible input values are called "state space". This is important to note that GOTCHA recommends test generation of partial functional specification because tool is not good in handling large models with large state space. A tester should add only important functions in GDL and restrict test state space by adding preconditions in actions [66].

GOTCHA-TCBeans does not provide GUI support to write machine specifications in GDL. A tester can use any editor for this purpose.

GOTCHA generates abstract test cases automatically from FSM which is generated from GDL. The generated abstract test suite, in XML for offline and online testing, contains information about: name of the model, list of state variables with their ranges, list of all

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

rules defined in GDL, and most importantly all abstract test cases with their inputs and expected outputs. Expected output value is used as test oracle [66].

A tester is required to write a testing interface between the model and the SUT. This interface is written using TCBeans classes. In addition to this, a translation table is also required which can be written in XML and/or by using TCBeans classes. The translation table is simply a mapping of calls to API methods of the SUT.

Test cases can be executed by an existing test execution engine or by the TCBeans. TCBeans comprise of two main parts. A2E (abstract to executable) test driver classes and A2C (abstract to concrete) test translator classes. When a tester wants to use available test execution engine, A2C translates the abstract test suite into script of concrete test cases that can be run by the execution engine. Abstract test cases are executed directly on the SUT using A2E classes. The A2E test driver is appropriate to test applications written in C, C++ and Java. Test execution traces could be viewed in TCBeans browser. GOTCHA-TCBeans provides model debugging facilities. Details are not available.

5.1.2 mbt

mbt is an open source model based testing tool implemented in Java programming language. It generates test sequences from FSM/EFSM and available on Tigris open source software engineering tools web site [18]. mbt has no GUI environment. It is used from command prompt and tester should know all commands and switches to activate different tool features.

mbt uses models described in GraphML[3] format. Tool does not support model creation and recommends yED[4] for this purpose. In yED created graph, vertices are states and edges are transitions.

Every state and transition of input model could have a multi-line label. Every state in model must always have a label and every model must have only one state with the 'Start'

[3] http://graphml.graphdrawing.org/
[4] http://www.yworks.com

label to indicate the starting state in the model. Every state and transition of input model may have one or more keywords. Keywords are reserved words which have special meanings. They are written within the label of state or transition in second line. For example, MERGE keyword is used by states when merging FSM/EFSM sub-models. State containing this key word is merged with the first occurrence of a state with the same label. REQTAG (=reqID) keyword is used for keeping track of which requirements are covered by the generated test sequence. This keyword is used to bind requirement IDs with the transitions. The first line of transition labels in FSM are the function names which should be called when this transition occurs. The name of method should start with "e_". Labels of transition in EFSM are used to add guard conditions. The format is:

Label Parameters [Guard] / Action1;Action2;....;ActionN;
Keyword

Transition label should start with "e_" and should not contain white spaces. This label indicates a method name which takes Parameters. Guard condition should return true to let corresponding transition occur. Actions are simply valid Java like statements.

States are treated as test oracle in mbt tool. Label of the state (except Start state) indicates a method name which should be called when that state is reached. Name should start with "v_". mbt generates code skeleton with these method names and tester is required to add necessary code to be used as test oracle. This generated code is used as test adapter.

mbt tool supports three algorithms to generate test sequences. The following are the details.

RANDOM: also called "Random Walk", randomly traverses un-visited transitions to generate test sequence.

A_START: generates the shortest possible test sequence. It is recommended only for small models. Bigger models might cause "out of memory" exception because shortest path calculation is not that much efficient and requires a lot of calculations.

SHORTEST_NON_OPTIMIZED: is a compromise between A_STAR and RANDOM. The algorithm works as follows:

1. Choose an transition not yet visited by RANDOM.
2. Select the shortest path to that transition.
3. Walk that path, and mark all those transitions as visited.
4. When reaching the selected transition in step 1, start all over, repeating steps 1 to 4.

Tester must indicate desired test generation method using corresponding name at command prompt or in parameters XML file.

mbt tool supports both online and offline testing. Online testing is partially automated because tester has to provide input values when test sequences are run. Because of this reason, mbt is not recommended for online testing. An offline test sequence is a sequence of transitions and states extracted from a test model by mbt. Each test cases has input and expected output values which are used as test oracle. mbt generates basic skeleton for test adapter class. The tool also provides options for partial states, transitions and requirements coverage. It also generates test sequence to cover only specific states, transitions and requirements.

mbt requires a template to generate test adapter in any programming language. Tester either could use existing template or could create his own template. The following template is used to generate test adapter class in Java programming language:

```
public void {LABEL}(){
        log.info( "{EDGE_VERTEX}: {LABEL}" );
        throw new RuntimeException( "The {EDGE_VERTEX}: {LABEL} is not implemented yet!" );
}
```

LABEL and EDGE_VERTEX are place holders which are replaced by the information from the test model. Tester further has to add code in adapter class to implement testing logic. Tester needs an appropriate test execution tool. In case of web application, Selenium[5] test tool could be used. The .class file of the test adapter class is copied to a specific location. Next, Selenium server is started and then mbt tool is invoked with an

[5] http://seleniumhq.org/projects/remote-control/index.html

XML file as parameter. The XML parameter file contains information about: locations of test adapter class and test model file, location of Selenium or other test cases execution server, test generation method and other values like percentage of edge or transition coverage. Other test execution tools recommended by mbt are HP QuickTest Professional[6] and IBM Rational Functional Tester[7].

5.1.3 MOTES

MOTES is a model based testing tool developed by ELVIOR[8]. This tool is an Eclipse plug-in and generates test case from EFSM model. MOTES does not provide test model creation and test execution facilities. It only generates test cases in TTCN-3 which can be executed by any TTCT-3 tests execution engine.

MOTES requires three input files to generates test cases. One file is the EFSM test model in XMI format. MOTES recommends Poseidon[9] and Artisan Studio[10] environments to create the UML state machine test model. This model should not have parallel regions to be treated as EFSM. The second file contains the test data (context variables) description in TTCN-3 and is created manually. Context variables are used in EFSM to store state information. This file is used to generate test input and expected output data to be used as oracle. The third file contains the interface specifications which define ports (methods to be executed on the SUT). These ports also define types of input parameters (from TTCN-3 test data input).

MOTES includes two test generation engines: (1) Model Checking engine and (2) Reactive Planning Tester engine. Model Checking engine uses Uppaal CORA[11] verification engine for test case generation which is available for Windows and Linux platforms. Uppaal CORA finds test sequences over the EFSM model. UML state machines do not have formally specified language for the presentation of guard

---

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

[7] http://www-01.ibm.com/software/awdtools/tester/functional/

[8] htpp://www.elvior.com/

[9] http://www.gentleware.com/

[10] http://www.artisansoftwaretools.com/products/artisan-studio/

[11] http://www.cs.aau.dk/~behrmann/cora/

conditions, input events and actions on transitions. Uppaal CORA needs this information in some formal language to generate test cases. MOTES uses a formal language MTL (MOTES transition language) in transitions of the EFSM. MTL details are available in [82].

Reactive Planning Tester engine does not generate test cases but it generates a reactive planning tester component that in test execution time generates tests case on-the-fly for online testing. The reactive planning tester is generated in form of TTCN-3 script. It mainly generates test cases for nondeterministic part of the SUT. In online testing, the reactive planning tester has to decide which move from the possible ones to take. For this purpose, it keeps track of executions of the SUT and chooses the appropriate execution path at the points of nondeterministic behaviour.

The generated test cases (from deterministic models) and the reactive planning tester (from nondeterministic models) are executed against the SUT using ELVOIR MessageMagic[12] TTCN-3 test tool. MessageMagic produces test reports in text and XML format. The reports contain information about verdict, and covered states and transitions in the model [82].

5.1.4 NModel

NModel is an open source model based testing tool for model programs written in C# and runs on Windows platforms [22]. Test cases are generated from model program generated FSM. This framework is developed at Microsoft Research and known as base of Spec Explorer [27]. NModel requires .NET framework to be installed on user machine. It provides several components for different testing activities and class library for writing model programs.

NModel consists of four components: mpv (Model Program Viewer), mp2dot (Model Program to Dot), otg (Offline Test Generator) and ct (Conformance Tester or Test Runner).

---

[12] http://www.elvior.com/messagemagic/general

A model program is defined by C# classes in a namespace. Each class consists of actions (possible state transitions) and variables. Variables declared in model programs are called state or context variables which can be as simple as an integer and as complex as collection of objects, list, map and bag [76]. Actions are methods in model program that are labeled with the [Action] attribute.

Classes defined in model program or namespace may be labeled with [Feature] attribute. A feature is a group of class variables and actions and can be given a name using [Feature("feature_name")]. The default feature name is the class name. Features are used to selectively include or exclude classes from a model program for exploration. Composition combines separate model programs into a new large model called the product that is explored automatically [76]. Each model program should have a factory method (called Create) as entry point to the model program for exploration.

NModel provides APIs to be used in model program. Model program can be analyzed to check consistency before the start of exploration process.

mpv performs the exploration and generates an FSM/graph from one or more model programs. The tool runs from command prompt and displays FSM in a GUI. It also provides facility to watch state variables. mpv supports machine verification and has ability to identify unreachable states. It uses Microsoft Automatic Graph Layout[13] (MSGL) to display graphs.

The mp2dot is another option to perform exploration and generate graph. It does not provide GUI and saves generated graph in PostScript which can then be displayed by any PostScript viewer.

The otg tool generates an offline test suite that achieve link coverage (all transitions) criterion of FSM. The syntax of generated text cases is similar to C# but it is not exactly C#. NModel uses this representation for internal use.

---

[13] http://research.microsoft.com/en-us/downloads/f1303e46-965f-401a-87c3-34e1331d32c5/default.aspx

The ct execute the test cases on the SUT. Tester is required to creates a test adapter (test harness) in C# to couple the SUT with ct. ct also performs online (on-the-fly) testing and can execute test cases on any implementation language.

5.1.5 TestOptimal

TestOptimal is a model based testing tool developed in Java programming language. It is a client/server web application and can be used in Windows and Linux environments. TestOptimal is developed to test enterprise web applications written in Java, Ruby, PHP, Perl and Groovy programming languages, however, it can also be used to test desktop applications. It supports both online and offline testing. It generates test sequence in many testing languages including TTCN and Python for offline testing [32].

Models in TestOptimal is graph which is EFSM where vertices are states and edges are transitions. Model in TestOptimal are created either using interactive HTML client or by WebMBT Builder. WebMBT Builder is used for web applications only. A tester can create reusable sub-models and then include them in the main model. Models created in GraphML[14], XMI[15] and GraphXML[16] formats can be imported to TestOptimal. The model could be tested using "Dry Run" facility provided by TestOptimal. By clicking on "Dry Run" button, tester instructs the tool to test the model. An error message is shown if an inaccessible state or transition is found.

TestOptimal provides 15 XML based tags with many attributes for writing scripts to test applications. These tags collectively called mScript. Tester uses mScript to create its testing logic. For example, tester can use "code" attribute of "action" tag to call some server side method. Further, he can instruct TestOptimal when to execute that action. Consider the following XML snippet of mScript XML file:

```
<state id="save_state">
  <transition event="save_text">
        <script type="action">
              <action code="$saveText('some_params')"/>
```

[14] http://graphml.graphdrawing.org/
[15] http://www.omg.org/technology/documents/formal/xmi.htm
[16] http://strategoxt.org/Transform/GraphXML

```
        </script>
  </transition>
</state>
```

It instructs TestOptimal to call saveText method with ”some_params” when "save_text" (which is transition id) event is fired when system is in state "save_state". ”some_params” are the data values sent to server from some HTML fields. "assert" tag of mScript can be used to define oracle when method returns some value. A tester can write test oracle logic using attributes of "assert" tag. TestOptimal server throws MBTException if assert does not hold and so test case fails.

Guard condition of EFSM can be implemented using "if" tag of mScript. Loops, server side logging, database operations, methods for state events (onentry, onexit) and pre/post transitions conditions can be added using mScript.

A tester can test parts (or sub-models) of the test model by using mCase. mCase is a group of states and transitions which tester wants to test. The states and transitions need not to be consecutive and tester can follow any order to add them in mCase. TestOptimal automatically generates the shortest path to cover all states and transitions. mCase are executed by mCase specific algorithms (see details below).

After creating the model, writing the test logic in mScript and creating mCase(s), the next step is to generate and run test cases.

TestOptimal provides 5 different algorithms to generate test case. A tester should select one before clicking "Run" to start testing process.

- **RandomSequencer:** generates test cases by randomly selecting a model transition.
- **GreedySequencer:** similar to RandomSequencer but avoids already covered (tested) transition. This algorithm considerably reduces test suite size, compared to RandomSequencer.
- **OptimalSequencer:** traverses all transitions to achieve 100% transition coverage of the model. Test suite size could have many test cases.

- **mCaseOptimal:** generates test suite to cover all transition and/or states in mCase. This algorithm is used to partially test the system by using tester defined mCase(s).
- **mCaseSerial:** same as mCaseOptimal but follows the order specified by mCase. The order of mCase is the order of selection of states and transitions.

TestOptimal provides information about covered and uncovered states and transitions with total number of traversals and test running time.

The vendor of TestOptimal maintains a wiki page for online documentation, and also provides online support. However, there is no online forum for this tool where the users' community could discuss their experiences and issues.

TestOptimal uses plug-ins to drive web applications during testing process. For example, it uses Selenium[17] to drive web application under test. Another option is HtmlUnit[18]. These plug-ins provide APIs which could be used to invoke pages, fill out forms, click links, etc.

TestOptimal also supports testing Java applications. Tester defines label to generate method names in test adapter class. TestOptimal automatically generates test adapter class skeleton from test model. Test adapter class should inherit com.webmbt.plugin.JavaTestPluginHandler class. Context variables are defined in test adapter class and MBTException is used to implement test oracle. To deploy, compile the test adapter class and copy .class file to /bin folder of TestOptimal folder to run/test the model with the TestOptimal browser [32].

5.1.6 Spec Explorer 2010

Spec Explorer 2010 (referred as Spec Explorer in rest of this document) is a model based testing tool from Microsoft that extends Visual Studio. It provides facilities to create and validate models, explore and generate test cases from them. Test models in Spec Explorer

[17] http://seleniumhq.org/
[18] http://htmlunit.sourceforge.net/

are model programs written in C#. Although, .NET programs can easily be tested, Spec Explorer can be used to test software written in any programming language.

The main UI components of Spec Explorer are Exploration Manager, Modeling Guidance, View Definition Editor, and Exploration Graph Viewer. Exploration Manager is the main control panel that provides support to validate and explore model program and generate test cases from them. Modeling Guidance feature helps creating machine for testing. Exploration Graph Viewer displays generated graph (FSM/ASM) when exploration is performed. Generated graph might have hundreds or even thousands of states. Exploration Graph Viewer also provides support to explore selected parts of graphs with zoom facility [27].

Model program consists of Rules (tagged with [Rule]) and state attributes. Rules or model methods determine what transitions can be produced leading from the current state in the model program. Actions, in simple words, are those methods which a tester would like to test. Each Rule have an associated Action that specifies the action invocation (transition). Each action invocation is called Exploration Step and collection of exploration steps is called Trace (test case). Since rules exist to specify what transitions the model program allows from a given state, the true output of a rule is zero or more transition labels representing possible next actions (transitions) from that state.

Actions are declared in configuration files which are created in Cord scripting language (.cord file). Cord is used to create Configuration for optimizing model exploration. A configuration can also include a set of switches and parameters to control exploration and testing. For instance, a switch can be used to avoid state space explosion (infinite states).

Spec Explorer explores a model program in order to generate a representative finite subset of the potentially infinite behavior of the FSM/ASM [68]. Spec Explorer requires machine to generate test model from model program. Machines are units of exploration, written in Cord and are used to perform an exploration of the associated model program. It can be a combincation (composition) of exisiting mahiches. A smaller machine (sub-model) can be created using slicing. Spec Explorer provides many slicing techniques to avoid state explosion. Some of them are: parameter selection, state filtering, and

behavioural restriction using trace patterns. Slicing requires tester intervention and skills, and could reduce test coverage [27].

Exploration Manager generates test cases in C# to execute them on the SUT. Spec Explorer supports two approaches to execute generated test cases on the implementation of the SUT. They can either be connected directly or through a test adapter. Connecting them directly means to declare the actual methods of the SUT in a Cord configuration. Consequently, the generated test code will call directly into the SUT. This option can only be applied to managed code implementations (written in .NET). For unmanaged code, a test adapter is required. In this case, adapter methods are declared as actions in Cord configuration files instead referring directly to the SUT. As a result, the generated test code will call methods in the adapter. The adapter concretizes test cases before executing them on the SUT and make verdict by compairing returned value against the expected output [27].

Execution results of test cases are displayed in Explorarion Manager. A test case can be debugged using Visual Studio debugger program.

5.1.7 AGEDIS

AGEDIS (Automated Generation and Execution of Test Suites for DIstributed Component-Based Software) is a set of integrated tools for behaviour modeling of distributed systems, test generation, test execution and result analysis [73]. It is developed by a research consortium of seven industrial and academic bodies with the aim to improve automation of software testing.

AGEDIS suite includes: a modeling tool, a test suite editor and browser, a test case simulation and debugging tool, a test coverage analysis tool, defect analysis and test execution report generator. All these tools are integrated through a graphical environment. There are other system components not visible to AGEDIS user they are: test model compiler, test generator engine and test execution engine.

AGEDIS is mainly developed in Java programming language and available for free. It runs on Windows and Linux platforms.

The behavioural model of the SUT is specified by AGEDIS Modeling Language (AML). AML is based on UML 1.4 and fully complaint with its syntax, however, it uses only a subset of UML diagrams which includes Class, State machine and Object diagrams. Each class diagram in test model has associated state machine describing behaviour of its object. Object diagrams are used to describe initial states of objects and hence of the SUT.

Class diagrams are used to capture the structural abstractions of the SUT. They provide AGEDIS user defined data types, attributes and constants (enumeration). The interface between the SUT and a class is described by attaching stereotypes <<controllable>> or <<observable>> to artefacts that could be a method or attribute associated with class. Controllable methods are those which change the state of the SUT when they execute. They do not return value. Observable methods return value when are executed on the SUT. Similarly observable attributes are used to store values returned from the SUT [72].

The class diagrams, object diagrams and state machines are annotated with the IF (Intermediate Format) action language. A tester can define variables, procedures, conditional and loop statements in IF. Guard conditions of state machine transitions are also annotated in IF and its format is "Trigger [guard] / Action". Blocks of IF code are attached with state machines using <<action>> stereotype.

Each construct of AML is transformed to IF before model is used to generate test cases. In other words all classes, objects and state machines are encoded into IF before the start of test generation process [79]. AGEDIS uses integrated Objecteeing[19] UML Modeler for modeling purposes. Test model is a mix of diagrams and IF action scripts. A tester provides three pieces of information to generate test cases: (1) test model, (2) the test execution directives, (3) test generation directives.

The Test Execution Directives (TED) describe the testing interface to the SUT. It is described in XML and composed using integrated Spy editor[20]. The XML document contains information about model translation. The model translation information

[19] http://www.objecteering.com/
[20] http://www.altova.com/xml-editor/

comprises mappings from the abstractions of data types, classes, methods, and objects described in the model to those of the SUT.

Test generation directives consist of important information which test generator needs to generate test cases. This information includes: coverage criteria, test constraints mainly using stereotypes. Each diagram in test model has its dedicated optional test constraints mentioned using stereotypes. For example, <<start>> and <<end>> are stereotypes for object diagram which are used to mentioned start and end states of an expected test case.

All the input information is compiled into IF and then passed on to test generation engine to generate test suite in XML format. Each generated test case contains test input and expected output. Output information is used as test oracle. Test cases can also be parameterized so that single test case could be used for many input values. This considerably reduces size of test suite.

AGEDIS offers only offline testing. A tester can use model simulator to study generated test suite. Simulator provides feedback by executing generated test suite on the test model by using message sequence charts. The embedded debugger can be used to debug a test case. Test execution engine runs generated test suite on the SUT. The test case concretization process is done using TEDs. The responses are compared with the expected output to make a verdict. Test executing engine records test execution trace which could be seen in AGEDIS editing tool. This tool helps tester to add or edit test information manually.

AGEDIS is able to execute test cases on the SUT developed in Java, C and C++ programming languages. Its test execution engine (Spider) is able to run test cases on distributed systems in heterogeneous environments.

AGEDIS provides coverage analyzer and defect analyzer tools for analyzing test suite execution. The coverage analyzer tool provides information about possible input values which are not covered and methods which are not invoked by the generated test suite. It provides guidelines to tester to cover uncovered input values and methods by generating coverage reports. The defect analyzer provides information about failed test cases. It

reads the test suite execution trace and helps tester to perform the failure analysis. This tool also groups test cases with respect to similar failure reasons which helps tester to fix problems in short time.

5.1.8 ParTeG

ParTeG (Partition Test Generator) is model based test generation tool developed at Fraunhofer Institute for Computer Architecture and Software Technology[21]. This tool is an open source Eclipse plug-in that runs on Windows and Linux operating systems.

ParTeG test model is called coupled model that comprise of UML 2.0 class and associated state machine diagrams. Coupled model is created using TopCased[22] UML Eclipse plug-in. Coupled model is annotated with OCL expressions which connect state machine to class diagram by using guard conditions of transitions (with calls to class operations). ParTeG does not support OCL collections in script. Format of the guard condition is Event [Guard] / Action. Any transition in state machine can refer to attributes and operations of coupled class using events, guards and actions of guard conditions.

OCL expressions of coupled model are used to generate test input value partitions. Input value partitions are then used during test execution to know possible violations of OCL constraints in the SUT. ParTeG generates test cases from OCL expression of coupled model. It constructs a transition tree to investigate possible tree paths. Each path in transition test tree correspond to a path (test case) generated from state machine. The OCL expressions of each path in the tree are transformed into conditions on input values. These conditions correspond to input value partitions for logical test cases. Values near the boundaries of these partitions are automatically selected as input values for concrete test cases where expressions of a path are used as test oracle [101].

ParTeG generates test cases in Java which are executed on the SUT using JUnit. Test cases could be debugged using integrated debugger of Eclipse environment. ParTeG does not support online test execution and model validation. There is no online support and

[21] http://metrik.informatik.hu-berlin.de/
[22] http://www.topcased.org/

forum available for this tool. This is a prototype tool and available free on sourceforge website[23].

5.1.9 Qtronic

Qtronic is developed by Conformiq [4] in Java programming language and runs on Windows and Linux. It has three main parts: Qtronic Computation Server, Qtronic Modeler and Qtronic Client. User has to load model on server to generate test cases. Qtronic Modeler is used to create test models using UML state machines. Test cases are generated from these models. Qtronic Modeler has a rich graphical environment with drag & drop and zoom facilities. Qtronic Client is available as Eclipse plug-in and standalone desktop application. A tester has choice to select one of both. Qtronic Client provides users with the facilities to create test model (using integrated Qtronic Modeler), select test coverage criteria using Coverage Editor, and analyze model and test suite execution results when they are run on the model. Qtronic creates test suites using different coverage criteria.

The test model in Qtronic is a combination of UML state machines and blocks of an object oriented programming language called QML (Qtronic Modeling Language). QML is a superset of Java and C# . The use of Qtronic Modeler to create UML state machines is optional. A model can be described using notations in XMI format. QML provides sufficient object oriented constructs to describe test model, however it does not include standard libraries that come with Java and C# programming languages. Transition String (guard condition) is used in state machine to attribute model. Format of Transition String is "<trigger> [guard] / <action>". Trigger is used to model the reception of an event. A guard specifies a condition for the transition to fire. Action specifies statements to perform when transition fires. An action could be a simple statement in QML, a conditional statement in QML, a call to method declared in QML or combination of all these. The functional requirements of the specifications that are selected for testing in the implementation are annotated on the model of the SUT using the keyword "requirement". Requirement keyword is written in Action part of the Transition String. Further details on QML are available in [41].

Qtronic generates input and expected output values from the test model written in QML. These input and output values become part of generated test suite in which output values act as test oracle. Qtronic provides facility to execute generated test suite on the test model to evaluate model coverage.

Qtronic can import models created in IBM Rhapsody[23] and Enterprise Architect[24]. Before generation of test cases, test model is loaded on Qtronic Computation Server. Qtronic Computation Server coverts QML blocks of code and (textual notation of) UML state machines into LISP and then abstract test suite is generated using different algorithms [39]. Qtronic does not provide integrated facility to execute test cases. Tester can export test cases in different formats like: C, C++, Java, Perl, Python, TCL, TTCN-3, XML, SOATest[25], Excel, HTML, Word and Shell Scripts. Qtronic generated test suites could be saved using version manager tools.

Online testing is done from within Qtronic Client environment by directly connecting to the SUT using a DLL (dynamic link library) plug-in interface and test execution engine. In this mode, test case generation and execution take place in parallel [75].

5.1.10 Test Designer

Test Designer is a model based testing tool included in Smartesting Center solution suite [29]. It is developed by Smartesting in Java programming language and runs on Windows and Linux operating systems. Test Designer has been used to test smartcards, electronic transaction, telecom, security, automotive control, banking and resource management software domains. It take UML 2.x model as input and uses test coverage criteria to generate abstract test cases and uses test adapters to translate abstract test cases into executable test scripts.

Smartesting Center provides automated bidirectional traceability between requirements and generated test case [40] by using IBM Rational DOORS (MS Word and Excel) as

[23] http://www-01.ibm.com/software/awdtools/rhapsody/
[24] http://www.sparxsystems.com.au/
[25] http://www.parasoft.com/jsp/solutions/soa_solution.jsp;jsessionid=aaa7EAlvipZLSd?itemId=319

requirements management tool and integrated HP Quality Center[26] (HPQC) web based application as test cases repository. Each requirement is defined and assigned a unique ID which is subsequently used for bidirectional traceability. HOQC automatically loads requirements using proprietary interfaces[27]. Smartesting Center uses IBM Rational Software Modeler[28] (RSM) for model creation. Smartesting Model Checker and Simulator are integrated with RSM. Other options for model creations are IBM Rational Software Architect[29] (RSA) and Borland Together[30].

Test models are developed using UML 2.x Class, State machine and Object diagrams. Class diagram defines the static view of the model. A tester can leverage the benefits of associations, enumerations, attributes and operations to define static abstractions and their dependencies of the SUT [56]. Object diagrams are used to define the initial state of the SUT by using objects and initial values of their attributes. Dynamic behaviour is captured using state machine. State machine is associated with the main test class of the model. Name of the main class should be annotated with <<SUT>> stereotype to increase model understanding. Other stereotypes can be used with class diagram to associate additional meanings to class methods. For instance, <<events>> are those actions on the SUT which are used in the associated state machine to trigger transitions between states, <<operations>> are those class methods which are called from action part of the transitions of state machine. Names of all methods defined in main test class are later used to generate test scripts.

Guard conditions of the transitions are formalized in OCL using Event[Guard]/Action format. Event is method of the main class, Guard is an OCL predicate and Action could be a method of the main class or an OCL postcondtion. Each part of Guard condition is optional. Action part of guard condition can be tagged with requirement ID using "--- @REG: requirement_id" tag. Test Designer generates traceability links using this information. Test model can be verified using Smartesting Model checker integrated with

---

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

[27] http://www-01.ibm.com/support/docview.wss?uid=swg24024729

[28] http://www-01.ibm.com/software/awdtools/modeler/swmodeler/

[29] http://www-01.ibm.com/software/awdtools/architect/swarchitect/

[30] http://www.borland.com/us/products/together/index.html

RSM/RSA. Model is exported to Test Designer tool to generate test cases and requirements-to-test cases traceability links. A test case composed of the following parts [56]:

- **Preamble:** which is a sequence of (zero or more) operations to reach the behaviour to be tested. In other words, a preamble sets the context for test case.
- **Body:** Expected results which are used as test oracle.
- **Postamble:** The sequence of operations to return to initial state of the SUT. This is an optional part and should be generated only when there is a need to initialize the SUT before executing new test case.

Test Designer provides the facility to execute generated test cases against the model to study model coverage. It provides seamless integrated support to publish test cases to different targets not only as test scripts in a test environment (such as HPQC) but also as HTML, XML, and as Java classes for JUnit. HPQC generates test script (test adapter) from test suite generated by Test Designer to execute them on the SUT.

Smartesting Center offers only offline testing. Company provides supports to users however no online forum could be found to discuss problems and experiences. The evaluation version of Test Designer is not available. Very limited information is available and that too is not enough to provide information needed by Systematic Review Criteria. Most of information about this tool is provided by Bruno Legeard (CTO Smartesting).

5.1.11 Rhapsody Automatic Test Generation

Rhapsody Automatic Test Generator (Rhapsody ATG) is a UML based test case generation tool developed by IBM [42]. It uses UML model developed in IBM Rhapsody and offers test case generation capabilities to perform structural and requirements based coverage. ATG extends Model-Driven Development to include both Model-Driven Code Generation and Model-Driven (model based) Test Generation. It automatically generates test cases to cover design, and generated code. ATG analyzes both UML models and the generated C++ code to automatically generate sets of model-based test cases. ATG does not generate test cases if the generated code is not provided in compiled form. It

generates test cases for model (states, transition, functions) coverage criterion and C++ code (statement, MC/DC) coverage criteria.

ATG enables Rhapsody users to generate test cases for single classes (unit testing), or for a set of classes (integration testing). The SUT must be represented as an executable component. This means that to generate test cases for the SUT (either a single class or a set of classes), the SUT must be compiled and linked into an executable.

Test cases consist of input stimuli, and expected output (oracle) as computed from a given model. ATG generates test cases that can be exported from Rhapsody in order to execute them with IBM TestConductor [42].

IBM TestConductor automatically generates test monitors and test drivers from Rhapsody sequence diagrams (SDs) to execute test cases on the SUT. During automated test execution, the generated monitors determine whether the executed model satisfies the selected sequence diagrams. ATG supports regression testing.

ATG supports requirements-based test coverage analysis to assess which requirements have been tested, and a structural coverage analysis to assess the degree of structural code coverage.

5.1.12 Rational SDL and TTCN Suite

IBM Rational SDL and TTCN Suite (SDL-TTCN) is used for software development of real time and communication systems. The SUT is described using the Specification and Description Language (SDL) in a graphical or textual format and test cases can be generated using TTCN. The suite runs on Windows, Solaris and Linux operating systems.

The SDL suite provides the SDL editor which is a powerful GUI environment to create system models. It supports creating high level to detailed level system features.

The model of an SDL system consists of a set of extended finite state machines (EFSM) that run in parallel. The Message Sequence Chart (MSC) language is used to describe the dynamic behaviour (traces) of the SDL model. The user can also create the expected dynamic behaviour of the SDL model using MSCs to test them against the SDL model using the SDL Explorer [45]. A

tester can also add Object Model Notation (OMN, similar to UML class, object, and state machine) diagrams in the SDL model to describe types, their initialization and dynamic behaviour.

The SDL Suite contains many graphical editors to provide rich GUI environment to develop SDL models. The SDL Editor is used for creating and editing specifications and descriptions using the SDL notations. It also provides validation and simulation features. The user can set break points and can simulate selected parts of the model. The OM Editor supports adding OMN/UML notations in the SDL model.

The SDL Suite allows adding C/C++ code in the model to demonstrate partial behaviour through simulation. The added code can use libraries to call external routines and can be executed during the main SDL model simulation.

After C/C++ code addition and simulation, the user can generate C/C++ deployable code from the SDL model. The SDL suite provides more than one compiler options to compile the generated code [45].

The SDL Suite is linked to the TTCN test generator framework through the Autolink feature of SDL Explorer. This provides an integrated SDL/TTCN environment to access SDL specification to generate TTCN test cases.

Test cases are generated from simulation traces. Each trace is generated with the help of the SDL Simulator and stored as an MSC. The Autolink configuration (with TTCN Suite) defines possible signal parameters (inputs), types conversions and state space characteristics (limits on input values) and the user can select the desired coverage percentage of MSCs (test cases). The TTCN Suite performs the state space exploration (the reachability analysis) before generating the test cases. Autolink provides a special state space exploration technique, called Tree Walk, which is optimized for finding paths through the SDL specification to achieve the desired coverage [44].

The SDL suite offers the Coverage Viewer which is a graphical tool that shows how much of a system has been covered during a simulation in terms of executed transitions and SDL model elements/components. By checking the system coverage, the user for instance can see what parts of the system have not been executed in the simulation.

The SDL and TTCN Suite supports integration with IBM DOORS to allow easy synchronization of the SDL model with system requirements. This facility allows traceability between requirements, SDL model, MSCs and TTCN tests cases.

5.2 Results

In this section, we present the results of our systematic review. We briefly described each tool in section 5.1. Although all selected tools are state based MBT tools, they vary in some aspects. Some tools use specification generated test models to create test cases. GOTCHA-TCBeans, NModel and Spec Explorer fall in this category. Some use a model diagram with a scripting language to generate test cases. TestOptimal and Qtronic are examples of these tools. There are also some tools which use more than one diagrams with scripting language to generate test cases. They are: AGEDIS, Test Designer and ParTeG. Our study further shows that all of these tools have subtle variations. Spec Explorer APIs are much bigger than NModel. Qtronic provides more coverage criteria than TestOptimal and MOTES. Test Designer is much more flexible, easy to use and more automated than AGEDIS and ParTeG. Further details are beyond the scope of this thesis.

All selected tools provide more than one coverage criteria to generate test cases and some claim to provide 100% coverage. The information provided in this section is based on the tool documentation (papers, manuals, cases studies). Although we ran some samples on selected tools to get their basic knowhow but we believe an in-depth comparison requires lot more work which is out of scope of this thesis.

A "Y" in states coverage should not always mean that the tool is capable to always provide 100% coverage. We strongly believe that there are many scenarios when full coverage is not possible. For instance when model is large and have many states and transitions.

5.2.1 Comparison 1: Test Criteria

This comparison is based on model flow coverage criteria discussed in section 4.2.

Table 1 shows the adequacy criteria supported by the selected tools. All tools except SDL-TTCN fully support transitions coverage. MOTES, NModel, SDL-TTCN and Spec Explorer do not support state coverage. A user can create scenario in MOTES by selecting states and transitions but generated test cases are based on selected transitions and not on selected states [82]. MOTES also supports transition pair coverage as Qtronic and Test Designer do. Sneak path or implicit consumption, All Paths and Parallel Transitions criteria are only supported by Qtronic. Implicit consumption/sneak paths criterion is not applicable to GOTCHA-TCBeans, NModel, and Spec Explorer because test models (FSM/ASM) in these tools do not have guard condition. Parallel transitions criterion is not applicable to GOTCHA-TCBeans, mbt, MOTES, TestOptimal, NModel and Spec Explorer. These tools use FSM, EFSM and ASM models to generate test cases and there is high level of complexity involved in using parallel behaviour in these models. MOTES, TestOptimal, SDL-TTCN and Test Designer support creation and coverage of test scenarios while ParTeG and Qtronic do not support this criterion. Test Designer has a feature called "scenario-based testing" which allows tester to simulate scenario before generating test cases. SDL-TTCN creates scenarios using MSC to generate test cases. mbt and AGEDIS partially support this coverage criteria because they allow creating scenarios by indicating start state, end state, and forbidden states. The tester does not have the freedom to define coverage order of transitions or states in a scenario. This criterion is not applicable to GOTCHA-TCBeans, NModel and Spec Explorer because these tools generate test models from specifications and do not provide graphical symbols to create models.

Table 1 – Model flow AC (adequacy criteria) comparison.

Tool Name	States	Transitions	Transition Pair	Sneak Paths	All Paths	Parallel Transitions	Scenarios
GOTCHA-TCBeans	Y	Y	N	N/A	N	N/A	N/A
mbt	Y	Y	N	N	N	N/A	P
MOTES	N	Y	Y	N	N	N/A	Y
NModel	N	Y	N	N/A	N	N/A	N/A
Spec Explorer	N	Y	N	N/A	N	N/A	N/A
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	N	N	Y
Rhapsody ATG	Y	Y	N	N	N	N	N
SDL-TTCN	N	N	N	N	N	N	Y

Table 2 shows the script flow comparison. Interface/Function criterion refers to the coverage of those selected methods which are called on the SUT. Test suite should have at least one occurrence of these functions. NModel, Spec Explorer, AGEDIS, Qtronic, Test Designer and Rhapsody ATG support function coverage. Model program in NModel supports coverage of functions (action) defined in the model program, however test cases are generated for transitions where label represents functions calls [76]. Similarly, Spec Explorer generates test cases to cover all functions (actions) defined in Cord file [27]. Both these tools assure 100% coverage of functions (actions) so they do not provide information about those functions which are not covered. In this case, Interface/Function criterion measures adequacy (AC). AGEDIS generated test suite aims to cover all selected methods (section 5.1). Its Defect Analyzer tool informs about those methods which are not covered by the generated test suite [72]. In Qtronic, methods are defined in QML script and tool supports method coverage criterion [41]. Test Designer generates test cases to cover methods defined in class diagram of test model. Names of these methods are also called keywords. Interface/Function criterion in AGEDIS, Qtronic, Test Designer and Rhapsody ATG is a coverage criterion (CC) because coverage information is generated by executing test cases on state machine models. There is a notion of methods or functions in GOTCHA-TCBeans, mbt, MOTES, ParTeG and SDL-TTCN but they do not support method (interface/function) coverage.

There is a notion of statement and control flow structure in scripting languages mScript (TestOptimal) and IF (AGEDIS), 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, 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. 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[31] (except atomic condition). Rhapsody ATG generates test cases for statement and MC/DC coverage for the generated C++ code which is white box testing, which is not considered in our comparison. There is a notion of statement (written in C/C++, see section 5.1.12 for details) in an SDL model but SDL-TTCN does not generate test cases to cover statements. ParTeG covers only Decision/ Branch criterion for simple OCL predicates and do not support Condition and MC/DC criteria on internal actions [101]. Test Designer support Decision/Branch, Condition, atomic/multiple condition and MC/DC criteria for guard conditions and internal actions [50]. ParTeG does not support atomic/multiple condition criterion. Qtronic is the only tool which generates test cases to cover statements in model as well as in script written in QML. Statement coverage criterion is not applicable to GOTACH-TCBeans, NModel and Spec Explorer because they generate FSM/ASM from model specifications. This criteria is also not applicable to mbt, ParTeG and Test Designer because their test models do not contain statements. Decision or Branch, Condition, MC/DC and Atomic/multiple condition coverage criteria are not applicable to those tools which generate models from specifications (i.e. GOTCHA-TCBeans, NModel, and Spec Explorer). It is also evident from Table 2 that mbt, MOTES, TestOptimal, AGEDIS, Rhapsody ATG and SDL-TTCN do not support script flow criteria. mbt and MOTES have no scripting language so these criteria are not applicable to them. TestOptimal, AGEDIS, and SDL-TTCN do no not support script flow coverage criteria although these tools have scripting languages. Qtronic does not support MC/DC criterion, however, it covers rest of criteria for model and QML script.

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

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

Tool Name	Interface/ Function	Statement		Decision or Branch		Condition		MC/DC		Atomic/ Multiple condition	
		AC	CC	AC	CC	AC	CC	AC	CC	AC	CC
GOTCHA-TCBeans	N	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
mbt	N	N/A	N/A	N	N/A	N	N/A	N	N/A	N	N/A
MOTES	N	N/A	N/A	N	N/A	N	N/A	N	N/A	N	N/A
NModel	Y (AC)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Spec Explorer	Y (AC)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	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	Y	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	Y	Y	Y	Y	Y	Y	Y
Rhapsody ATG	Y(CC)	N	N	N	N	N	N	N	N	N	N
SDL-TTCN	N	N	N	N	N	N	N	N	N	N	N

Table 3 shows that GOTCHA-TCBeans, NModel, Test Designer, Spec Explorer and Qtronic generate test cases for data (adequacy) criteria. The first four 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. GOTACHA-TCBeans, NModel and Spec Explorer generate test model from model specifications (model programs) and tools cover One value and All values criteria by generating many (although not infinite) states. However, all these tools strongly recommend to put limit on input values (state space) using different approaches. Tester imposes limit on state space while writing instructions using relational operators in GDL for GOTCHA-TCBeans. NModel and Spec Explorer provide C# classes for this purpose which a tester can use in model programs to put bounds on state space. Spec Explorer, Qtronic and Test Designer support boundary value coverage. Spec Explorer provides classes to achieve boundary value, Pair-wise and N-wise coverage through rules defined in model programs. Qtronic and Test Designer provide GUI support to select data coverage criteria. mbt, MOTES, AGEDIS, ParTeG, Rhapsody ATG and SDL-TTCN do not support data coverage. AGEDIS generates parameterized test cases, which could be used to pass different input values through a test adapter, the tool itself does generate test cases to cover data input values.

Table 3 – Data and requirements coverage comparisons.

Tool Name	One value	All values	Boundary values	Pair-wise	Requirements
GOTCHA-TCBeans	Y	Y	N	N	N
mbt	N	N	N	N	Y (CC)
MOTES	N	N	N	N	N
NModel	Y	Y	N	N	Y (AC)
Spec Explorer	Y	Y	Y	Y	Y (AC)
TestOptimal	N	N	N	Y	N
AGEDIS	N	N	N	N	N
ParTeG	N	N	N	N	N
Qtronic	N	N	Y	N	Y (CC)
Test Designer	Y	Y	Y	Y	Y (CC)
Rhapsody ATG	N	N	N	N	Y(CC)
SDL-TTCN	N	N	N	N	N

Table 3 also indicates that six tools support requirements criterion. This is mainly achieved by defining unique identifiers (IDs) for requirements and then attaching them to test model transitions. NModel and Spec Explorer allow the tester to attach requirements IDs in the model to make sure that all requirements are covered from the generated FSM/ASM models. mbt, Qtronic, Test Designer and Rhapsody ATG also measures the coverage of requirements defined in the test model.

5.2.2 Comparison 2: Tool Automation Criteria

Table 4 indicates that most of MBT tools rely on third-party (TP) tools to perform related activities. TestOptimal, Qtronic and SDL-TTCN provide full support for test model creation using purpose build modeling environment. AGEDIS, ParTeG, Test Designer Rhapsody ATG and Spec Explorer provide seamless (S) integration with the third party tools while the user of GOTCHA-TCBeans, mbt, MOTES and NModel have to import models created externally. Note that GOTCHA-TCBeans, MOTES and NModel require input scripts, created using any text editors. Spec Explorer extends Visual Studio to create the model while Rhapsody ATG uses Rhapsody model through seamless integration. These clearly indicate that instead of investing time and money in creating modeling support, most vendors prefer to use existing environments.

Eight tools support model verification. Most of the tools which do not execute test cases (F in Table 1, Part II) do not support test case debugging. ParTeG uses Eclipse while Spec Explorer extends Visual Studio for this purpose. mbt, NModel, TestOptimal, Test

Designer and Spec Explorer support sub modeling which means these tools facilitate testers in creating complex and large test models (programs). mbt, TestOptimal and Test Designer integrate graphical symbols (classes, state machines, FSMs, EFSMs) of test models while NModel and Spec Explorer allow to compose model programs to generate larger FSM/ASM test models (composition).

Only Test Designer fully supports end-to-end requirements traceability thanks to its integration with a requirement management tool, namely DOORS (Part II). mbt, Qtronic and Spec Explorer support requirements coverage but only partially support traceability (no requirement management tool). Rhapsody ATG and SDL-TTCN partially support requirements traceability. Both these tools use DOORS to link the test model with textual requirements but there is no traceability support from test cases to the model. Similarly to modeling, most MBT tools rely on third party tools for test case execution (with the exception of GOTCHA-TCBeans, AGEDIS, NModel, Spec Explorer and SDL-TTCN). Details on test cases execution environment/tools can be found in tool description in section 5.1. All selected tools generate test cases.

Only Qtronic supports regression testing (i.e. the notion of retestable, reusable and obsolete test cases). We could not find this information for Test Designer.

Table 4 – Automation coverage comparison

Part I

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

Part II

Tool Name	Test Case Generation	Test Case Execution	Requirements Traceability	Regression Testing
GOTCHA-TCBeans	F	F	N	N
mbt		TP-D	P	N
MOTES		TP-E	N	N
NModel		F	N	N
Spec Explorer		F	P	N
TestOptimal		TP-S	N	N
AGEDIS		F	N	N
ParTeG		TP-S	N	N
Qtronic		TP-E	P	F
Test Designer		TP-S	TP-S-F	?
Rhapsody ATG		TP-S	TP-S-P	N
SDL-TTCN		F	P	N

5.2.3 Comparison 3: Test Scaffolding Criteria

Support is provided for adapter creation (Table 5). When there is support, it is either partial (P), which means only a skeleton of functions is created automatically, or full, which means the tester do not need to create adapters. Only ParTeG and SDL-TTCN generate complete test adapters. ParTeG is a prototype research tool which generates test cases to test only Java applications while SDL-TTCN is a commercial tool and test C/C++ applications. Test Designer and Rhapsody ATG seamlessly integrate with third party tools for this. Similarly, the oracle creation is partially supported by most of the tools, and the same comments apply to ParTeG and Rhapsody ATG. No tool is able to generate stubs, which indicates that tester needs to do extra work in adapter layer to emulate undeveloped system components.

mbt partially supports (P) online testing because it prompts the tester for input values during the test execution process. MOTES, TestOptimal and Qtronic use third party tools for offline and online testing. Online testing can be done by linking Qtronic with a test execution engine through a DLL plug-in while MOTES embed online testing component in test suite so no linking of test generation tool is required. TestOptimal uses components like Selenium to perform online testing while GOTACHA-TCBeans and NModel provide full support for online testing. Test Designer, Spec Explorer, Rhapsody ATG and SDL-TTCN do not support online testing. GOTCHA-TCBeans, NModel, Spec Explorer, AGEDIS and SDL-TTCN provide full support for offline testing. All other tools use third party tools for this purpose.

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	N	N/A	TP-E	TP-E
NModel	P	P	N	F	F
Spec Explorer	P	P	N	N	F
TestOptimal	P	P	N	TP-S	TP-S
AGEDIS	P	P	N	N	F
ParTeG	F	F	N	N	TP-S
Qtronic	N	N	N/A	TP-S	TP-E
Test Designer	TP-S-P	TP-S-P	N/A	N	TP-S
Rhapsody ATG	TP-S-P	TP-S-P	N/A	N	TP-S
SDL-TTCN	F	F	N	N	F

5.2.4 Comparison 4: Miscellaneous Criteria

Miscellaneous criteria appear in Table 6. Six of the tools use (E)FSM/ASM, five use UML (class, or/and object diagram, and state machine diagrams) while one uses SDL model. Spec Explorer generates test cases from FSM/ASM. Note that all tools except GOTCH-TCBeans and Test Designer are available for evaluation. GOTCHA-TCBeans is available to IBM internal users and AGEDIS members. mbt, ParTeG, NModel and AGEDIS are free. mbt and NModel are open source which means their code can be used to develop a customized tool provided tools' license allow this. NModel was created at Microsoft Research and currently not supported. Spec Explorer is free but the user has to buy Visual Studio to use it. Most of these tools can be used to test all types of software

applications. Rhapsody ATG is specialized in testing of embedded systems while SDL-TTCN is good for real time systems.

Five of tools are able to execute tests on any target platform or programming language. These tool generate test cases in more than one programming and scripting languages like C/C++, Java, TTCN, HTML, XML, TCL and many more. GOTCHA-TCBeans, AGEDIS and ParTeG generate language specific test cases. But libraries and wrappers are available to execute these test cases on any platform. However, tester has to provide extra binding logic in test adapter layer. Spec Explorer and NModel execute test cases on .NET managed code written in any .NET enabled programming language.

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
NModel	FSM	Open source	All	.NET
Spec Explorer	FSM/ASM	Commercial	All	.NET
TestOptimal	FSM/EFSM	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
Rhapsody ATG	UML	Commercial	Embedded	C++
SDL-TTCN	SDL	Commercial	Real time	C/C++

6 RELATED WORK

Most of the primary study 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 [85]. Its focus therefore differs from ours (research/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 [81], the authors were 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, i.e., "a broad review of primary study to identify what evidence is available"[81], rather than a systematic literature review.

Surveys of some MBT tools are discussed in [54, 95]. Again, these are not systematic reviews since no systematic procedure to identify primary study (relevant tools) is described: and indeed some of our tools are not discussed in [54, 95]. 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 [20, 49, 71, 99, 100]. 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.

7 VALIDITY OF RESULTS

State-based MBT testing tools are the primary study of this systematic review. We collected, studied and analyzed more than 328 research papers and related technical reports and selected 78 support materials to develop our understanding of MBT. Our search process (section 3.2) describes the precise steps we followed in accordance with literature on systematic literature reviews.

In this section, we discuss three types of threats to validity of this systematic review. They are: Conclusion validity, Construct validity and External validity. Conclusion validity is about the validity of outcomes of our systematic review. Construct validity is concerned about the relation between research and implementation. External validity is about generalization of our results to cover other five paradigms (section 4.1).

7.1 Conclusion validity

The tools' list presented in section 3.5 is quite comprehensive but, despite all our efforts, we are not certain that all available state-based MBT commercial and research tools are included. There might be some tools which we could not find.

We provided results (section 5.2) in tabular form to indicate our findings. However, there are several limitations which a reader should have in mind. Although we tested some tools using sample models, most of our results are based on tools' documents and discussions with MBT experts. The "Yes-No-Partial" notion is not supported by a detailed qualitative comparison. A detailed qualitative analysis based on numerous case study models is required to discuss qualitative results. Our results should not be used to extract qualitative measures to compare selected tools nor do we declare that some particular tool is superior to others based on some criterion. For example, the fact that two tools support transition coverage does not mean the quality of the generated test suites, for instance in terms of execution time, size or achieved structural coverage is the same. Similar arguments are valid for other test criteria. Further, the fact that two tools fully support test case debugging does not mean they provide equal GUI and functional

support to track and analyze test case failure reasons. Similar arguments are valid for other evaluation criteria.

7.2 Construct validity

Our systematic review criteria (Chapter 4) were developed after studying support (research) material. They contains all parameters (criteria) which we find important for state-based MBT tool analysis. We analyzed test case generation, automation support and scaffolding capabilities of tools. We made every effort to avoid biased selection of review criteria but we also believe that there is always room for improvements, which will require additional studies and analysis of support material. One of the objectives of this systematic review is to help identify gaps between research and tool support in the MBT area (section 3.1). One needs to study all of available research material and understand all available state-based MBT tools to be able to identify gaps. Despite of all our efforts, we do not claim that we studied everything related to state-based MBT and tools, and the list of gaps we identified is complete.

7.3 External validity

MBT researchers have grouped modeling notations into six paradigms. We provided systematic review of state-based MBT tools. Our results should not be generalized to access MBT tools for other five paradigms. We believe that similar systematic reviews should be conducted to evaluate tools which use the test model developed in other modeling notations.

8 CONCLUSION AND FUTURE WORK

Model based testing (MBT) which, in this document, 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 [99]. 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 [85]). 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 [81]) identified 30 tools and 2 APIs, we evaluated 15 tools and compared 12. 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.

There is a lot of room for improvements with respect to model flow, script flow, data and requirements, tool automation, and test scaffolding comparison criteria. Most of the tools do not support Transition Pair, Implicit Consumption or Sneak Path, All Paths and Parallel Transitions coverage criteria. Similarly in script-flow criteria, a number of tools (e.g. mbt, MOTES, TestOptimal and AGEDIS) do not support Decision/Branch, Condition, MC/DC and Atomic condition coverage criteria. All these criteria are quite important in testing of embedded software used in safety critical software. Moderate support is provided for data coverage criteria but none of the selected tools support data flow coverage criteria as described in [65, 99]. We also believe that support for test scaffolding can be improved. Adapter and stub creation, and Oracle automated construction require improvement as no tool fully supports these. Automation criteria show that most of the MBT tools use third party support for model creation, Test case

debugging and Test case execution which we believe is more reliable because of choice available in selecting mature and reliable tools.

The main contributions of this research work 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.

We described all seven MBT types of notations (paradigms) in Chapter 4 and covered state/transition in this thesis. We evaluated those state-based MBT tools which comply to our systematic review criteria described in Chapter 4. In our future work, we would like to extend this activity to cover the other six paradigms to evaluate the tools which we omitted in this study. For this we will need to extend our test coverage criteria (section 4.2) while other comparison criteria seem adequate enough to be used for all paradigms. However, at the same time, we are keen to find and add any automation or test scaffolding parameters (criteria) which we find important for MBT tool support. We would also like to perform case studies to qualitatively analyse and compare MBT tools.

REFERENCES

[1] AGEDIS Project. http://www.agedis.de/. (Last accessed April 2010)

[2] Agile Modeling, http://www.agilemodeling.com (Last accessed March 2010)

[3] ATD-Automated Test Designer. http://geekswithblogs.net/srkprasad/archive/2004/09/07/10748.aspx. (Last accessed March 2010)

[4] Conformiq Qtronic. http://www.conformiq.com/. (Last accessed April 2010)

[5] CTesk. http://www.unitesk.com/. (Last accessed December 2009)

[6] D-MINT Project, http://www.d-mint.org/ (Last accessed April 2010)

[7] ELVIOR LLC., http://www.elvior.ee/ (Last accessed April 2010)

[8] Errfix. http://code.google.com/p/errfix/. (Last accessed January 2010)

[9] The Escalator Tool, http://heim.ifi.uio.no/~massl/escalator/ (Last accessed March 2010)

[10] Expecco. http://www.exept.de/en/products/expecco. (Last accessed June 2009)

[11] GATeL. http://www-list.cea.fr/labos/gb/LSL/test/gatel/index.html. (Last accessed June 2009)

[12] IBM GOTCHA-TCBeans. http://www.haifa.ibm.com/projects/verification/gtcb/index.html. (Last accessed April 2010)

[13] JavaTesk. http://www.unitesk.com/. (Last accessed December 2009)

[14] JUMBL. http://sqrl.eecs.utk.edu/esp/jumbl.html. (Last accessed December 2009)

[15] KeY. http://www.key-project.org/. (Last accessed July 2009)

[16] MaTeLo. http://www.all4tec.net/. (Last accessed July 2009)

[17] The MathWorks, http://www.mathworks.com/ (Last accessed April 2010)

[18] mbttigrisorg. http://mbt.tigris.org/. (Last accessed April 2010)

[19] Microsoft Developer Network, http://msdn.microsoft.com (Last accessed April 2010)

[20] Model Based Testing by Zoltan Micskei, http://home.mit.bme.hu/~micskeiz/pages/modelbased_testing.html (Last accessed April 2010)

[21] ModelJUnit. http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/. (Last accessed April 2010)

[22] NModel. http://www.codeplex.com/NModel. (Last accessed May 2010)

[23] ParTeG. http://parteg.sourceforge.net/. (Last accessed May 2010)

[24] PrUDE. http://www.isot.ece.uvic.ca/prude/. (Last accessed December 2009)

[25] Reactis. http://www.reactive-systems.com/. (Last accessed April 2010)

[26] SIEMENS Corporate Research, http://www.usa.siemens.com/en/about_us/research.htm (Last accessed December 2009)

[27] Spec Explorer 2010. http://msdn.microsoft.com/en-us/devlabs/ee692301.aspx. (Last accessed May 2010)

[28] Spec Explorer Forum, http://social.msdn.microsoft.com/Forums/en-US/specexplorer/threads (Last accessed May 2010)

[29] Test Designer. http://www.smartesting.com/. (Last accessed May 2010)

[30] Test Suite Generation/Reduction (TSGR), http://www.site.uottawa.ca/~ural/TSGR/ (Last accessed June 2010)

[31] TestEra: The MulSaw Project, http://projects.csail.mit.edu/mulsaw/ (Last accessed June 2010)

[32] TestOptimal. http://testoptimal.com/. (Last accessed May 2010)

[33] TGV. http://www.irisa.fr/pampa/VALIDATION/TGV/. (Last accessed December)

[34] Time Partition Test (TPT). http://www.piketec.com/products/tpt.php. (Last accessed December 2009)

[35] TorX. http://fmt.cs.utwente.nl/tools/torx/introduction.html. (Last accessed April 2010)

[36] T-VEC Simulink Stateflow Tester. http://www.t-vec.com. (Last accessed April 2010)

[37] Uppaal-TRON. http://www.cs.aau.dk/~marius/tron/. (Last accessed April 2010)

[38] Wikipedia the free encyclopaedia, http://en.wikipedia.org (Last accessed June 2010)

[39] Conformiq Qtronic SG: Semantics and Algorithms for Test Generation, 2008

[40] Smartesting Center: Test Generation and Requirements Traceability, White Paper, 2008

[41] Conformiq Qtronic User Manual, 2009

[42] *Rhapsody Automatic Test Generation Add On*, IBM Rational, 2009.

[43] *SDL and TTCN Suite User Guide*, IBM Rational, 2009.

[44] *SDL and TTCN Suite User Manual*, IBM Rational, 2009.

[45] *SDL Getting Started Guide*, IBM Rational, 2009.

[46] Ammann P. and Offutt J., *Introduction to Software Testing*, Cambridge University Press, 2008.

[47] Becker P., "Model-based testing helps Sun Microsystems remove software defects," *TechRepublic online magazine*, 2002.

[48] Beizer B., *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Edition, 1990.

[49] Belinfante A., Frantzen L. and Schallhart C., "Tools for Test Case Generation," in M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems*, Springer, pp. 391-438, 2005.

[50] Bernard E., Bouquet F., Charbonnier A., Legeard B., Peureux F., Utting M. and E. Torreborre, "Model Based Testing from UML Models," *Proc. Lecture Notes in Informatics (INFORMATIK 2006)*, pp. 223-230, 2006.

[51] Bertolino A. and Marchetti E., "Introduction to a Reasonably Complete and Coherent Approach for Model-Based Testing," *Electronic Notes in Theoretical Computer Science 116 Elsevier*, pp. 85-97, 2005.

[52] Binder R. V., *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Object Technology, Addison-Wesley, 1999.

[53] Bishwal B. N., Nanda P. and Mohapatra D., "A Novel Approach for Scenario-Based Test Case Generation," *Proc. IEEE International Conference on Information Technology*, pp. 244-247, 2008.

[54] Boberg J., "Early Fault Detection with Model-Based Testing," *Proc. ACM SIGPLAN workshop on ERLANG*, pp. 09-20, 2008.

[55] Borger E. and Stark R., *Abstract State Machine: A Method for High Level System Design and Analysis*, Springer-Verlag, 1st Edition, 2003.

[56] Bouquet F., Grandpierre C., Legeard B. and Peureux F., "A Test Generation Solution to Automate Software Testing," *Proc. International Conference on Software Engineering*, pp. 45-48, 2008.

[57] Braspenning N. C. W. M., Mortel-Fronczak J. M. and Rooda J. E., "A Model-Based Integration and Testing Method to Reduce System Development Effort," *Elsevier*, vol. 164, pp. 13-28, 2006.

[58] Broy M., Jonsson B., Katoen J.-P., Leucker M. and Pretschner A., *Model-Based Testing of Reactive Systems*, Lecture Notes in Computer Science, 1st Edition, 2005.

[59] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2nd Edition, 2004.

[60] Campbell C., Grieskmp W., Nachmanson L., Schulte W., Tillmann N. and Veanes M., "Model-Based Testing of Object Oriented Reactive Systems with Spec Explorer," MSR-TR-2005-59, 2005.

[61] Denaro G., Polini A. and Emmerich W., "Early Performance Testing of Distributed Software Applications," *Proc. ACM SIGSOFT Software Engineering Notes*, pp. 94-103, 2004.

[62] Dyba T., Kitchenham B. and Jorgensen M., "Evidence-based software engineering for practioners," *IEEE Software*, vol. vol. 22 (1), pp. 58-65, 2005.

[63] El-Far I. K., "Enjoying the Perks of Model-Based Testing," *Proc. the Software Testing, Analysis and Review Conference (STARWEST 2001)*, 2001.

[64] El-Far I. K. and Whittaker J., "Model-Based Software Testing," *The Encyclopaedia on Software Engineering, Wiley*, 2001.

[65] Fantinato M. and Jino M., "Applying Extended Finite State Machines in Software Testing of Interactive Systems," *Lecture Notes in Computer Science (Springer)*, vol. LNCS 2844, pp. 109-131, 2003.

[66] Farchi E., Hartman A. and Pinter S. S., "Using a Model-based Test Generator to Test for Standard Conformance," *IBM System Journal - special issue on Software Testing*, vol. 41(1), pp. 89-110, 2002.

[67] Gravell A. M., Howard Y., Augusto J. C., Ferreira C. and Gruner S., Concurrent Development for Model and Implementation, University of Southampton., 2003

[68] Grieskamp W., Gurivich Y., Schulte W. and Veanes M., "Generating Finite State Machines from Abstract State Machines," *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 112-122, 2002.

[69] Gronau. I., Hartman A., Kirshin A., Nagin K. and Olvovsky S., "A Methodology and Architecture for Automated Software Testing,", 2000, A Technical research report by IBM Haifa Research Lab.

[70] Hartig W., Habermann A. and Mottok J., Model-Based Testing for Better Quality, Vector Informatik GmbH, 2009

[71] Hartman A., "AGEDIS Model Based Test Generation Tools," AGEDIS Consortium, http://www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf, 2002.

[72] Hartman A., "AGEDIS Project Final Report," AGEDIS-1999-20218, 2004.

[73] Hartman A. and Nagin K., "The AGEDIS Tools for Model-Based Testing," *Proc. International Symposium on Software Testing and Analysis*, pp. 129-132, 2004.

[74] Heimdahl M., "Model-Based Testing: Challenges Ahead," *Proc. IEEE International Computer Software and Applications Conference*, pp. 330, 2005.

[75] Huima A., "Invited Talk: Implementing Conformiq Qtronic,", *Testing of Software and Communication Systems*, vol. 4581/2007, SpringerLink, pp. 1-12, 2007.

[76] Jacky J., Veanes M., Campbe;; C. and Schulte W., *Model-Based Software Testing and Analysis with C#*, Cambridge University Press, 1st Edition, 2008.

[77] Jonsson B., *Finite State Machine*, Model Based Testing in Reactive Systems, vol. LNCS 3472, Springer, 2005.

[78] Kalaji A., Hierons R. and Swift S., "Generating Feasible Transition Path for Testing from an Extended Finite State Machine (EFSM)," *Proc. IEEE International Conference on Software Testing and Verification and Validation*, pp. 230-239, 2009.

[79] Kirshin A., Cavaran A. and Trost J., "AGEDIS Modeling Language (AML)," AGEDIS, 2002.

[80] Kitchenham B., Dyba T. and Jorgensen M., "Evidence-based software engineering," *Proc. ACM/ International Conference on Software Engineering*, pp. 273-281, 2004.

[81] Kitchenham B. A., "Guidelines for Performing Systematic Literature Reviews in Software Engineering," Technical Report EBSE-2007-01, Keele University, 2007.

[82] Kull A., *Model Based Testing of Reactive Systems*, PhD Thesis, Tallinn University of Technology, Faculty of Information Technology, 2009

[83] Kwinkelenberg R. and Schoch J. P., *Smartesting for Dummies*, Wiley Publications, 2008.

[84] Lee D. and Yannakakis M., "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, vol. 84, pp. 1090-1126, 1996.

[85] 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.

[86] Neto A. and Travassors G., "Supporting the Selection of Model-Based Testing Approaches for Software Projects," *Proc. ACM/IEEE International Conference on Software Engineering AST*, pp. 21-24, 2008.

[87] Neto A. and Travassors G., "Surveying Model-Based Testing Approaches Characterization Attributes," *Proc. IEEE International Symposium on Empirical Software Engineering*, pp. 324-326, 2008.

[88] Paradkar A., "Case Studies on Fault Detection Effectiveness of Model-Based Test Generation Techniques," *Proc. ACM International Conference on Software Engineering A-MOST*, pp. 01-07, 2005.

[89] Patton R., *Software Testing*, SAMS, 2nd Edition, 2005.

[90] Pender T., *UML Bible*, Wiley Publishing Inc., 2003.

[91] Pretschner A., "Model-Based Testing in Practice," *Lecture Notes in Computer Science*, vol. LCNS 3582, pp. 537-541, 2005.

[92] Pretschner A. and Philipps J., *Methodological Issues in Model-Based Testing*, Model Based Testing of Reactive Systems, vol. LCNS, Elsevier, 2005.

[93] Robinson H., "Obstacles and opportunities for model-based testing in an industrial software environment,", 2000.

[94] Rocha C. R. and Martins E., "A method for model based test harness generation for component testing," *Journal of the Brazilian Computer Society*, vol. 14 (1), 2008.

[95] Saifan A. and Dingel J., "Model-Based Testing of Distributed Systems," School of Computing Queen's University Canada, 2008.

[96] Sims S. and Varney D., "Experience Report: The Reactis Validation Tool," *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 137-140, 2007.

[97] Stobie K., "Model-Based Testing in Practice at Microsoft," *Electronic Notes in Theoretical Computer Science (Elsevier)*, vol. 111, pp. 05-12, 2004.

[98] Tretmans J., "Model Based Testing with Labelled Transition Systems," *Formal Methods and Testing*, vol. Springer (4949), pp. 01-38, 2008.

[99] Utting M. and Legeard B., *Practical Model-Based Testing: A Tools Approach*, Morgan Kauffmann, 2007.

[100] Utting M., Pretschner A. and Legeard B., "A Taxonomy of Model-Based Testing," Department of Computer Science The University of Waikato 04/2006, 2006.

[101] Weißleder S. and Schlingloff B., "Automatic Test Generation from Coupled UML Models using Input Partitions," *Proc. International Conference on Model Driven Engineering Languages and Systems, CiteseerX*, pp. 23-32, 2007.

Appendix A MODELS FOR MBT

Selected state/transition based models are succinctly reviewed in this section. The provided information is not used by model based testing tools as is. Current MBT tools (support) use Finite State Machines (Section A.1), Extended Finite State Machines (Section A.2), Abstract State Machines (Section A.3), Simulink/State flow chart (Section A.4), UML state machine (Section A.5), Harel Statechart (Section A.6), and Labelled Transition Systems (Section A.7).

A.1 Finite State Machine

A finite state machine (FSM) contains a finite number of states and produces output on state transition after receiving input. There are two types of FSMs: Moore Machines and Mealy Machines. Mealy machines model FSM more properly and are more often used than Moore machines. A Mealy machine has finite number of states and produces outputs after receiving inputs [84]. In rest of this document FSM mean Mealy machine.

According to [77], an FSM could formally be defined as the following.

An FSM is a quintuple $M = (I, O, S, \delta, \lambda)$ where

- I and O are finite nonempty sets of input and output symbols respectively.
- S is a finite non-empty set of states.
- δ: $S X I \rightarrow S$ is the state transition function.
- λ: $S X I \rightarrow O$ is the output function.

At any state s $\in S$, an input "a" could produce an output λ (s, a) and transforms itself to a new state δ(s, a). An FSM can be considered a directed labelled graph where S is the set of vertices and I is the set of inputs. For each s $\in S$ and a $\in I$, there is an edge from s to δ(s, a) labelled by "a/b" where "b" is the output symbol λ (s, a). See the Figure 1, which is an example of FSM.

FSM in Figure 1 has four states s_1, s_2, s_3 and s_4. When system is in state s_1, an input "b", which produces output "0", triggers a self transition which does not change state of the system. If input is "a" then output is "0" and transition causes system to move to new state s_2. Similarly other two states are traversed.

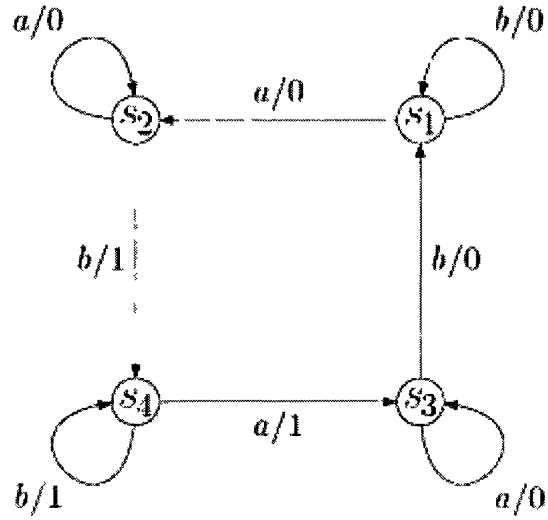

Figure 1 – Finite State Machine [84].

Applying an input sequence $x = a_1 a_2 ... a_k \in I$, starting in a state s_1 takes the machine successively to a sequence of states $s_2, s_3, ..., s_{k+1}$, where $s_{i+1} = \delta(s_i, a_i)$ for $i = 1, 2, ..., k$ and produces a sequence of outputs $b_1b_2...b_k \in O$ where $b_i = 1, 2, ..., k$. We extend the transition of output functions from input symbols to sequences of input symbols, by defining $\delta(s_i, x) = s_{k+1}$ and $\lambda(s_1, x) = b_1b_2....b_k$.

A sequence of states transitions from any initial state to any final state is called a Use Scenario [65]. An FSM can only model the control part of a system. An extension is needed in order to model a system which has control and data parts. Such systems are usually represented by Extended Finite State Machine (EFSM).

A.2 Extended Finite State Machine

The EFSM is a 6-tuple [78] $M_e = (S, s_o, V, I, O, T)$ where

- S is a finite set of logical states.

- $s_o \in S$ is the initial state.
- V is a finite set of internal variables.
- I is a set of input declarations.
- O is the set of output declarations.
- T is a set of finite transitions.

The transition $t \in T$ is represented by a 5-tuple [78] (s_s, i, g, op, s_e) where

- s_s is the start or initial state.
- $i \in I$ is the input which may have associated input parameters.
- g is a logical extension which is called guard.
- op is the operation such as output or assignment statements.
- s_e is the end state of transition t.

An EFSM uses a variable, called state variable, to represent the current state of the model. It could also have other variables called context variables which are used to save internal state information. A state transition occurs when machine takes a transition to change from one state to other state. Value in state variable is adjusted accordingly to exhibit new machine state. If a transition has a guard on context variables then guard must be satisfied in order for transition to take place. A transition might have one or more operations (op) to be executed when it is taken [78].

A.3 Abstract State Machine

Abstract State Machine (ASM) was introduced in mid 1980s by Yuri Gurevich. It is composed of states which are considered arbitrary data structures of types (primitive and composite). A sequential ASM is defined as set of transition rules of the form:

if *Condition* ***then*** *Updates*

It transforms first order structures (the state of the machine). The condition or guard under which a rule is applied is a variable free first order formula and Updates is a finite set of function updates (containing only variable free terms) of form [55]:

$f(t_1, t_2, t_3, ..., t_n) := t$

The notation of ASM run is the notion of computation of transition systems. An ASM computation step in a given state consists in executing simultaneously all updates of all transitions whose guards are true in the state.

A.4 Simulink/Stateflow Chart Machine

Simulink is a graphical data flow language, consisting of blocks which process data and signal lines which carry data between blocks [96]. It is used for modeling and simulation of systems. The Stateflow chart functions as an EFSM within a Simulink model. The stateflow machine is the collection of stateflow blocks in a Simulink model. The Simulink model and the Stateflow chart machine work seamlessly together. A simulation automatically executes both the Simulink blocks and Stateflow chart of the model. Stateflow charts consists of graphical and textual objects. See Figure 2 adopted from [17].

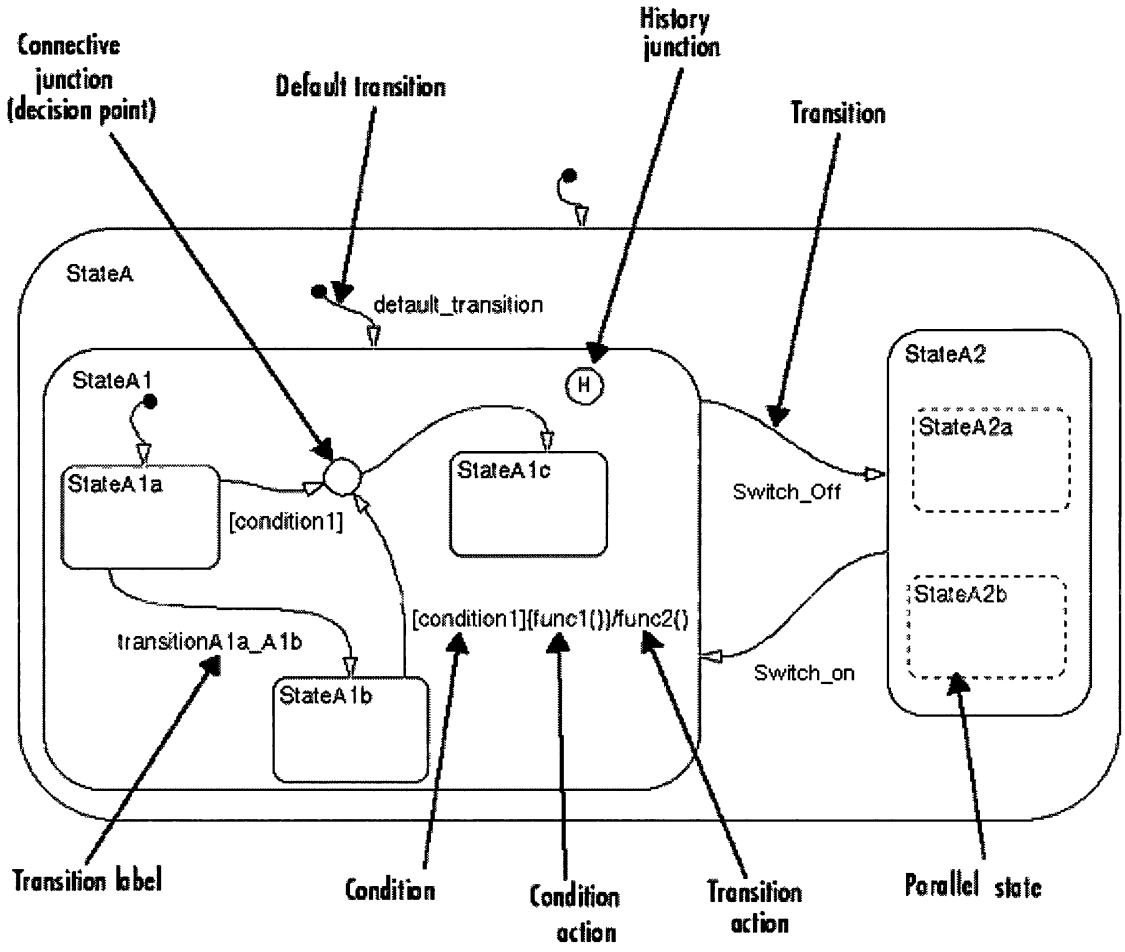

Figure 2 – Stateflow chart [17]

State

A state is the basic notation of Stateflow chart. It represents mode of the system. States could be parallel or exclusive. State could also be a Superstate and Substate. A superstate contains one or more substates. A substate is part of a superstate. In Figure 1, StateA is the superstate of substates StateA1 and StateA2. A superstate could have parallel states. Parallel states are used to model concurrent behaviour of a system. In Figure 2, StateA2a and StateA2b are parallel states of StateA2 substate.

Action

States can have actions that are executed in a sequence based upon the types of its actions. When a state is active, the Stateflow chart takes on that mode to indicate active

state. When a state is inactive, the Stateflow chart is not in that mode to indicate inactive state. The activity or inactivity of a Stateflow chart's states dynamically changes based on events and conditions. The occurrence of events drives the execution of the Stateflow diagram by making states become active or inactive. At any point in the execution of a Stateflow diagram, there is a combination of active and inactive states.

Transition

Behaviour of the model changes from one state to other state through an object called Transition. Transition usually links two states, a source state and a destination state. A transition might have an optional transition label. A label describes the circumstances under which the system moves from one state to another. In Figure 2, Switch_off and Switch_on are two transition labels. Default transitions specify which exclusive (OR) state is active when ambiguity exists among two or more exclusive (OR) states. In Figure 2, StateA1 is the default active substate when StateA is the active superstate.

Events

Events are non graphical objects and are not represented in a Stateflow chart. They drive the Stateflow chart execution. The occurrence of an event causes the status of the states to be evaluated. The occurrence of an event can trigger a transition to occur or can trigger an action to be executed. Data is another non graphical object (not shown in diagram) which is used to store numerical values for reference in Stateflow chart model.

Condition

A condition is a Boolean expression which could be True or False. A transition only occurs when condition (expression) returns true. In Figure 2, [condition1] represents a Boolean expression that must be true for the transition to occur.

Junctions

A stateflow char model could have history and connective junctions. A history junction records the most recently active state of a chart or superstate. If a superstate with

exclusive (OR) decomposition has a history junction, the destination substate is the substate that was most recently active. Connective junctions are decision points in the system. They are graphical objects which provide alternative ways to represent system behaviour. Actions take place as part of Stateflow chart execution.

The action can execute either as part of a transition from one state to another or based on the activity status of a state. Transitions ending in a state can have condition actions and transition actions [17]. In Figure 2, [condition1]{func1()}/func2() could be used to explain the order and execution of actions (func1() and func2() in this case). We assume that states StateA1b and StateA1c have direction transition and there is no connection junction or decision point between them. We further assume that initially an event is fired while StateA1b is in active more and condition condition1 is true which causes:

- Condition action func1() is executed and completed while keeping state StateA1b active.
- Any exit action of StateA1b is executed (there is no exit action in this case).
- State StateA1b is marked inactive.
- The transition action func2() is executed and completed.
- State StateA1c is marked active.
- Any entry action of StateA1c is executed and completed (there is no entry action in this case)
- Transition is complete.

A.5 UML State Machine

The UML state machine is a directed graph in which nodes represent states and connectors represent transitions [38]. It depicts the various states that an object may be in and the transitions between those states. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

States

A state represents the condition of an object which is represented as rounded rectangles labelled with state names. The details of object's condition could be examined by looking at values of object attributes and relationships of it with other objects or instances. An object may perform some work while staying in its present state. The transitions, represented as arrows, are labelled with the triggering events followed optionally by the list of executed actions. The initial transition originates from the solid circle and specifies the default state when the system first begins. Every state diagram should have such a transition, which should not be labelled, since it is not triggered by an event. The initial transition can have associated actions [38]. In Figure 3, Source State is the start state of transition and Target State is the end state of transition.

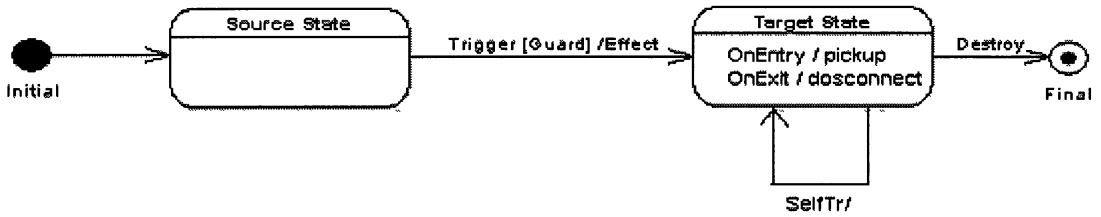

Figure 3 – A Simple UML state machine with start and end states

A state might have substates.

Trigger and Event

A trigger is cause of the transition. It could be an event, change in some condition or passage of time [2]. An event is anything that happens in a system. It could be a signal sent by some behaviour, for example: a key press, or mouse click. A trigger defines the types of events that can initiate a transition between states [90]. An event can have associated parameters which could carry some important information. Once generated, the event instance goes through a processing life cycle that can consist of up to three stages. First, the event instance is received when it is accepted and waiting for processing (e.g., it is placed on the event queue). Later, the event instance is dispatched to the state machine, at which point it becomes the current event. Finally, it is consumed when the

state machine finishes processing the event instance. A consumed event instance is no longer available for processing [38].

Transitions and Actions

Transitions model the movement from one state to other state. A transition is modeled by an arrow and caused by a trigger. Each transition could also be associated with an event. A transition originates from source state and ends at target or destination state. Some transitions might have same source and target state. Such transitions are called self-transitions. In Figure 3, the transition with trigger SelfTr is a self transition. A transition could also be the result of invocation of some method which introduce some changes in state of the object [90]. When an event occurs, state machine responses by performing some action. That action could be changing the value of a variable, calling a method and then might return some value, firing of another event to start some other action. The response action could do one or all things mentioned previously. A transition may have a trigger, a guard expression and an action with format like Trigger [Guard] / Action.

Guard Conditions

Guard conditions are Boolean expressions which return True or False on evaluation. A guard condition must return true in order to traverse a transition. A guard expression is usually written in OCL. However, a user could also use some programming language syntax (for example Java) or natural text to write guard condition [2]. Guard conditions are shown in square brackets; for example [conditionFlag == 0].

Substates

A state may have nested substates to design complex behaviour. Such state is called surrounding or superstate. A state machine could have more than one nesting level. Figure 4 precisely shows two level state machine of a washing machine. "Running" state is the supersatet of "Spinning" and "Washing" substates. If a machine is in the nested state, it is automatically is in corresponding superstate. If an event is fired in the context of substate which is not handled by it, it is automatically handled by superstate. If event is

not handled by any of machine's state, then it is simply discarded (i.e. implicit consumption).

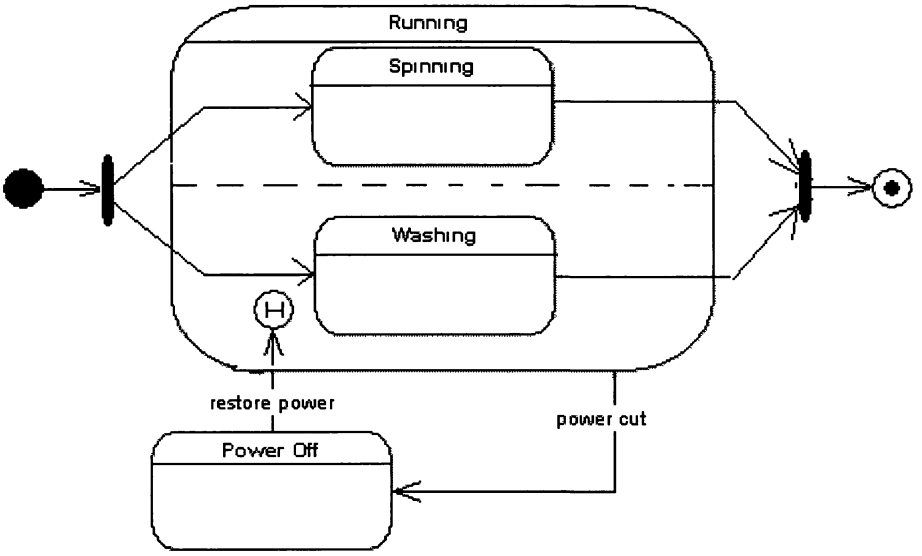

Figure 4 – A UML state machine with nested, history states and parallel regions

History State

A history state is useful to save context of state machine when it is interrupted. It is represented with an history state symbol (H-circle). Figure 4 shows that Running state has history state attribute which means that this state can save history of washing machine at the time of interruption. Washing machine goes to "Power Off" state when there is a power cut. When power is restored, machine enters into "History State" symbol which means it restores itself where it last left off [2].

Concurrent or Orthogonal Regions

A state may be divided into regions containing substates that execute in parallel. Such regions are called Concurrent or Orthogonal regions. Concurrent regions are used to design concurrently active parts of a system. Figure 4, "Spinning" and "Washing" substates belong to two separate concurrent regions.

A.6 Harel Statecharts

Harel statechart is ancestor of current UML state machines. It allows the modeling of states, superstates, substates, orthogonal regions and transitions with condition. In Harel statechart, a condition could evaluate to more than one value not just True or False as in case of UML state machines. A superstate in Harel state machine could have substates in orthogonal regions with AND and OR combinations. Harel statecharts work almost the same way UML state machines do. However, the UML state machines are more powerful than Harel statecharts. Further details and comparison between the UML state machine and Harel statechart is beyond the scope of this document.

A.7 Labeled Transition System

Labeled transition system (LTS) is a structure consisting of states with transitions, labelled with actions between them. The states model the system states and the labeled transitions model the actions that a system can perform. The transition systems are represented as graphs, where nodes represent states and labelled edges represent transitions. An LTS defines the possible interactions a system could have with its environment [98]. It is widely used to study properties of a reactive system. The following is the formal definition of LTS.

An LTS is a 4-tuple <Q, L, T, q_0> where:

- Q is a countable, non-empty set of states.
- L is a countable set of labels.
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation.
- $q_0 \in Q$ is the initial state.

If an LTS moves from state q to q' and corresponding transition is labelled as μ, then we indicate this transition as $q \xrightarrow{\mu} q'$, or the transition (q, μ , q') ∈ T. The transition (q, μ, q') indicates that when the system is in state q, it performs action μ and moves to state q'. Similarly, if system in state q' and performs action μ' to go to state q'' that is: $q' \xrightarrow{\mu'} q''$,

then these transitions could collectively represented as: $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$ and more precisely as: $q \xrightarrow{\mu\,\mu'} q''$. In general, the composition of transitions $q_1 \xrightarrow{\mu_1\,\mu_2\,\ldots\,\mu_n} q_2$ expresses that the system when in state q_1 can perform the sequence of actions μ_1, μ_2,..., μ_n, to move to state q_2 [98].

Figure 5 presents three examples of a labelled transition system representing a Candy Machine.

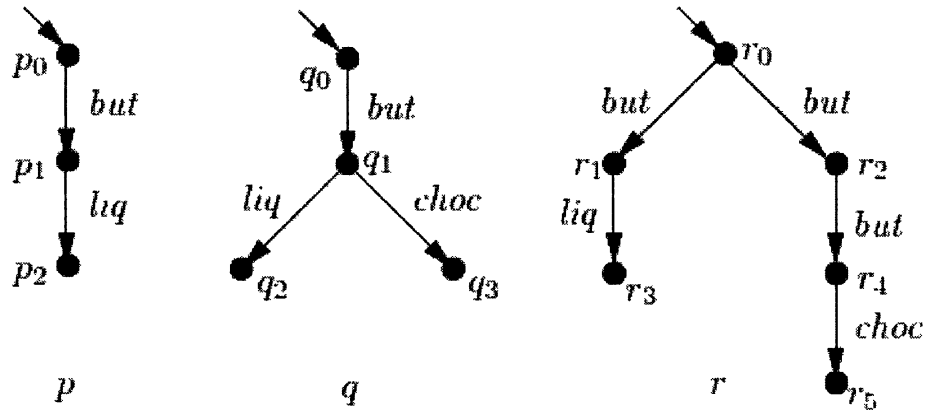

Figure 5 – Labeled Transition Systems of a sample Candy Machine, from [98]

There is a button interaction "but", and labels for chocolate "choc" and liquorice "liq". The dangling arrow points to the initial state. For machine p, we have that $p_0 \xrightarrow{but} p_1$ and also $p_1 \xrightarrow{liq} p_2$. Moreover, $p_0 \xrightarrow{but\ liq} p_2$. Similarly two compositions of transitions could be written for machine q due to its non-deterministic behaviour. For machine r, we have $r_0 \xrightarrow{but} r_1$ and also $r_0 \xrightarrow{but} r_2$. Similarly composition of two transitions for r is $r_0 \xrightarrow{but \cdot liq} r_3$ but $r_0 \xrightarrow{but \cdot choc} /$.

The labels in L represent the observable actions of a system. They model the system's interactions with its environment. Internal actions are denoted by the special label τ ($\tau \notin$ L), which is assumed to be unobservable for the system's environment. Also states are assumed to be unobservable for the environment [98].

A labelled transition system defines the possible sequences of interactions of a system which it could have with its environment. These interactions are abstract, in the sense that

they are only identified by a label. There is no notion of direction of the interaction and of input or output. LTS with simple labels are not feasible for testing purposes because it does not posses inputs and outputs.

Inputs and Outputs

An LTS with inputs and outputs is a 5-tuple <Q, L_I, L_U, T, q_0> where:

- < Q, L_I U L_U, T, q_0> is a labeled transition system in LTS (L_I U L_U).
- L_I and L_U are countable sets of input labels and output labels respectively where $L_I \cap L_U = \emptyset$

Inputs are usually denoted by "?" and outputs with "!". Input-output LTS is used to model a system with I/O behaviour. In these models, inputs are provided by the system's environment and never refused by the system while outputs are provided by the system and never refused by the system's environment which means all inputs are enabled in all states of an LTS.

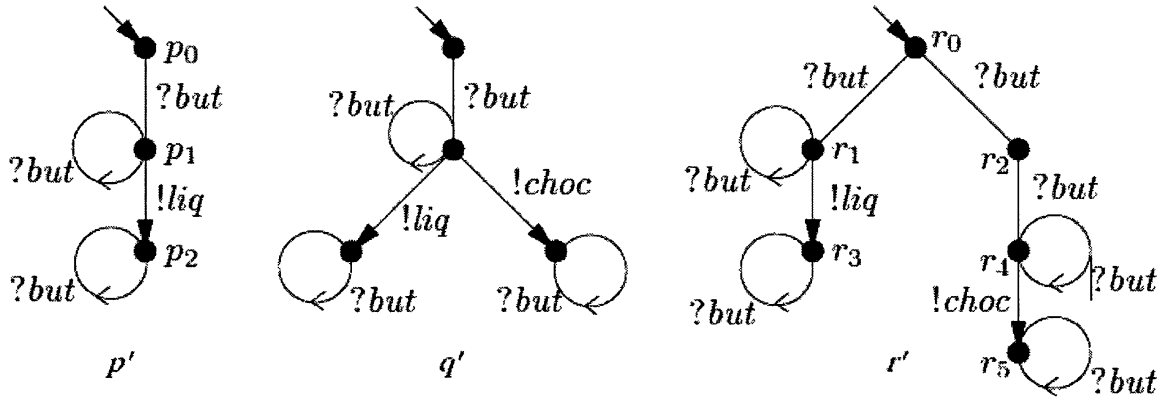

Figure 6 – Labeled Transition Systems of sample Candy Machine with inputs and outputs, from [98]

Figure 6 shows labeled transition system with input and output. It is the same LTS which is shown in Figure 5.. All states of p', q' and r' machines are enabled for all inputs.

Appendix B SEARCH STRINGS

This section lists search strings we used to search MBT tools for primary study and support material. Based on the research question described in section 3.1, we selected two relevant groups of terms which are population terms and intervention terms, and then further split intervention terms into two groups, **Error! Reference source not found.** and **Error! Reference source not found.**. Strings in **Error! Reference source not found.** are general MBT-related terms which we mainly used to search support material and tools information published in papers. **Error! Reference source not found.** are mainly names of tools and vendors which are used to search tool manuals, reports and white papers.

B.1 ACM Digital Library

- "model[32] based testing"
- "model driven testing"
- "model based testing with UML[33]"
- "model based testing with " + software[34]
- "model driven testing with" + software[35]
- "model based test coverage criteria"
- "model driven test coverage criteria"

---

[32] The word "model" is replaced by model, specification, requirements, UML (and all its diagrams), FSM, EFSM, Statechart, ASM, LTS, IOLTS, Simulink Stateflow chart and timed automata in the first two query strings.

[33] UML is replaced by FSM/EFSM, Statechart, LTS, IOLTS, Simulink Stateflow chart, ASM, time automata, SDL, use cases, activity diagrams, sequence diagram and state machine. We also used the full words strings like Unified Modeling Language for all acronyms. We could use OR operator but we found more material using short query strings.

[34] The term "software" is replaced one by one with tool, software, framework, API, application program interface and all strings in Intervention Terms2.

[35] The term "software" is replaced one by one with tool, software, framework, API, application program interface and all strings in Intervention Terms2.

- "model based testing using UML[32]"
- "model driven testing using UML[32]"
- "test coverage criteria of UML[32]"
- "UML[32] test criteria"

B.2 IEEE Xplore Digital Library and SpringerLink

- "model" and ("based" or "driven") and "testing"
- "model based testing" and (with or using) and ("UML" or "unified modeling language" or "FSM" or "finite state machine" or "EFSM" or "extended finite state machine" or "state machine" or "use case" or "activity diagram" or "SDL" or " Specifications Description Language " or " timed automata " or "LTS" or "labeled transition system" or "IOLTS" or "input output labeled transition system" or "ASM" or "abstract state machine" or "component diagram" or "collaboration diagram" or "sequence diagram" or "class diagram" or "object diagram")
- "model" and ("based" or "driven") and "testing" and ("tool" or "tools" or "software" or "framework" or "frameworks" or "API" or "application program interface")
- "model" and ("based" or "driven") and "testing" and ("with" or "using") and ("Qtronic" or "Reactics" or "Spec Explorer" or "TDE/UML" or "Smartesting" or "Conformiq" or "D-MINT" or "Test Designer" or "T-VEC" or "MaTelo" or "ATD-Automated Test Designer" or "errfix" or "GATel" or "mbt" or "ModelJUnit" or "NModel" or "ParTeG" or "Simulink Tester" or "Simulink/Stateflow chart" or "TestOptimal" or "TGV" or "Time Partition Test" or "TorX" or "expecco" or "PrUDE" or "KeY" or "JUMBL" or "UniTesk" or "Escalator" or "MOTES" or "AGEDIS" or "GOTCHA-TCBeans" or "Uppaal TRON" or "TestEra" or "exprecco" or "AsmL" or "IBM" or "Rhapsody" or "Tau"

or "Statemate" or "SDL" or "TTCN" or "suite" or "Automatic Test Generator" or "ATG" or "TestConductor")

B.3 ELSEVIER

Format and contents of the search strings for ELSEVIER are same as of IEEE Xplore.

B.4 Wiley InterScience

Format and contents of the search strings for Wiley InterScience are same as of ACM Digital Library.

B.5 EI Engineering Village

Format and contents of the search strings for EI Engineering Village are same as of IEEE Xplore.

B.6 CiteseerX

Format and contents of the search strings for CiteseerX are same as of ACM Digital Library.

B.7 Google and Google Scholar

Google and Google Scholar search engines offer many operators including "OR","+" and wild cards. We simply used ACM Digital Library search strings one by one for our search process.