Master Thesis

THE SURVEY OF MODEL BASED TESTING AND INDUSTRIAL TOOLS

by

Mohammad Mahdi Eslamimehr

LIU-IDA/LITH-EX-A-08/020-SE

April 2008

Linköping University
Department of Computer and Information Science

Master Thesis

THE SURVEY OF MODEL BASED TESTING AND INDUSTRIAL TOOLS

by

Mohammad Mahdi Eslamimehr

LIU-IDA/LITH-EX-A-08/020-SE

April 2008

Acknowledgments

Hereby, I would like to thank my supervisor and examiner Professor Kristian Sandahl for his guidance and help throughout my work. I also extend my sincere gratitude to Professor Mariam Kamkar for all her encouragements and benignities.

I would like to dedicate my thesis to my family, for their extreme support and patience during my years in Sweden at the University of Linkoping. I am indebted to them for every success I achieve in my life.

Mohammad Mahdi Eslamimehr
June 2008

Abstract

Model Based Testing is a software testing method in which test cases are entirely or partially originated from a behavioral model that expresses some facets of the *System Under Test*. Mostly, the basic model is abstract and tries to describe the system in whole or in less detailed mode. Thus, it is undisputed that the derived test cases from this abstract model should be functional and as abstract as the original model. Basically, we can call these test cases an *Abstract Test Suite*. It should be also emphasized that abstract test suites are not potentially executable since they are abstract and not on the actual level (they can only provide *Executable Test Suites*, to run against of *System Under Test*). *Executable Test Suites* are created by mapping function from the abstract test suites that is appropriate for testing. In contrast, as far as we are dealing with *Online Testing*, we have the abstract test suites only as a ground concept, not as an explicit artifact.

On other hand, the accuracy and precision of the results of Model based testing are hardly depend on the quality of underlying CASE tools which have been applied on the system under test. Variety of aspects such as: algorithms for test case generation, accuracy of oracle, and model coverage should be considered to end up with an appropriate tool for particular purpose. Conformiq Qtronic is one of the most credited CASE tools in MBT area. Even though the Conformiq Software is a young company, their tool receives much attention recently and became one of the first choices to many huge companies like Ericsson and Nokia.

Key Word: Model Based Testing, System under test, Model, Software requirements, Online testing, Executable test suite, Abstract test suite.

1. Introduction 7
1.1 Background of MBT 8
1.1.1 MBT Deployment 9
1.1.2 Algorithms for Test Case Generation 9
1.2 MBT Automation 11
1.3 Thesis Topics 12
1.4 Research Methods 12
1.5 Contribution 13
2. Theoretical Framework 15
2.1 An Argument about Modeling 15
2.2 Specification 16
2.3 MBT Fundamental Tasks 18
2.3.1 Understanding the SUT 18
2.3.2 Test Case Generation 21
2.3.3 Generating Expected Outputs 22
2.3.4 Running the Tests 22
2.3.5 Comparing expected output with real output 23
2.4 Key Observations 24
2.4.1 State Explosion 24
2.4.2 Test Coverage Parameter 24
2.4.3 MBT Advantage 25
2.4.4 MBT Obstacles 26
3. Tools for MBT 28
3.1 What is the MBT CASE Tool? 28
4. An Industrial Case: Conformiq Qtronic 36
4.1 Brief History of Conformiq Co. 36
4.2 Introduction to CQ 37
4.2.1 CQ Design and validation phases 37
4.2.2 CQ Testing Cost Assessment 37
4.2.3 CQ in Software Process 39
4.2.4 CQ Advantages 40

4.2.5 Anatomy of CQ 40
4.2.6 Design of CQ 42
4.2.7 Modeling in CQ 42
4.2.8 CQ Main Functionality 43
4.2.10 Test Case Generation 44
4.2.11 Scalability 46
5. Conformiq Qtronic Analysis 48
5.1 Reliability of the CQ: 48
5.2 Credibility of test case generation algorithms 49
5.3 Efficiency 51
Observation: We compared there different scale SUT from small system (Calculator) to semi-Large SUT (Inventory System). The results confirm what we expected from the CQ. Small system requires less time to create rather than the larger one. However, the larger system we have more test case we have, and that is a very good point of CQ. While we have a more complex system we need more test cases to check the functionalities of the system. The other important issue that uncovered during the efficiency evaluation is that the number and precision of the test case generated automatically is higher than the ones created manually. 52
5.4 Maintainability 52
5.5 Quality of Results 54
5.6 Portability 54
6. Future of MBT & Qtronic 57
6.1 Future of MBT 57
6.2 Remained Part of Qtronic 60
References 62

1. Introduction

This chapter introduces the fundamental thoughts behind the thesis. Furthermore, it summarizes methodology issues, the goals and contributions of the thesis, and also describes how it is organized.

Many research groups have studied *White-Box* or *Code-Based* testing until present. However, systems have become more sophisticated and code lines have grown incomparably than a decade ago [1]. For business systems, testers generally adopt *Black-Box* testing rather than *White-Box* testing, since it does not seem reasonable for testers to dive through details in the program code. It requires huge amount of effort, money, and time. On the other hand, White-Box testing is still used for critical software like rail and air traffic control.

In classical software engineering, up to 50% of the total development cost is related to testing [2]. This measurement also contains the cost of debugging, software testing is still assumed to be one the most conspicuous procedures in *Software Quality Assurance*. It provides a series of tasks to compare, and verify the system's real and expected behavior [3].

The expected behavior of the system is explained in specification documents. Therefore, the main activity is to create test cases based on such a specification. The notion of *Model Based Testing* (MBT) is to show expected behavior explicitly, in the form of a behavioral model. Once these models are derived to precisely express real requirements, traverse through the model can be offered as further test cases for respective implementation. The MBT concept is particularly noticeable, since it is

widely definite that besides the advantages of even automatic testing, only modeling can help to make the requirements clear. Moreover, executable models are usually so accurate at the level that they basically make prototypes [4].

1.1 Background of MBT

Model Based Testing is a software testing method in which test cases are entirely or partially originated from a behavioral model that expresses some facets of the *System Under Test*, (SUT). Mostly, the basic model is abstract and tries to describe the system in whole or in less detailed mode. Thus, it is undisputed that the derived test cases from this abstract model should be functional and as abstract as the original model. Basically, we can call these test cases an *Abstract Test Suite,* ATS. It should be also emphasized that abstract test suites are not potentially executable since they are abstract and not on the actual level (they can only provide *Executable Test Suites*, the ETS, to run against of SUT). ETS are created by mapping function from the abstract test suites that is appropriate for testing. In contrast, as far as we are dealing with *Online Testing*, we have the abstract test suites only as a ground concept, not as an explicit artifact [5].

Many methods have been offered to generate test cases from models. It is not possible to choose a method as the best way to create test cases, since fundamentally software testing often is heuristic based and experimental. Most of the time the package is created, namely *Test Requirements*, which includes the test stop conditions and information regard to the SUT part which should be tested. Test requirements are usually a result of merging the whole test origins related to the design decision. Figure 1 shows the general architecture of a MBT system and the relation between the parts of this framework. Figure 2 represents an example of a work flow for model based testing. It starts with formulating the requirements and ends up with comparing the actual and expected results by an oracle.

Model based testing usually assumes as a shape of black box testing because the test suites are totally generated from models, except the source code. Here we can say this is not precise claim because MBT can be joined to source code level test coverage measurement [6]. Furthermore, it is possible to provide a functional model by means of existing source code.

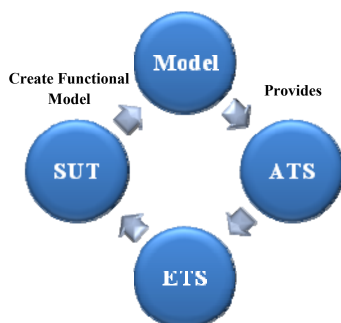


Figure 1: General Architecture for MBT

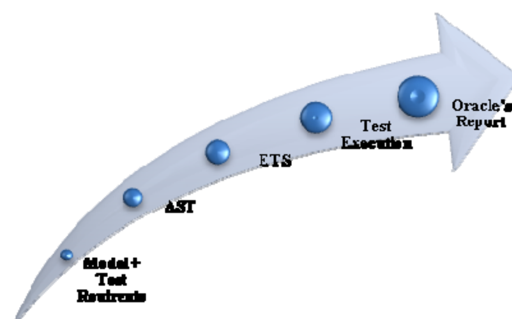


Figure 2: example of work flow for MBT

Basically, in Model Driven Engineering, models are designed and created in parallel with the development of the SUT. On the other hand, it is also possible to provide the model after the system is completed. Models are generally produced manually.

1.1.1 MBT Deployment

Several methods have been offered to deploy MBT: *Offline Generation of Executable Tests* [7], *Offline Generation of Manually deployable Tests* [8]*, and Online Testing* [9].

In the offline generation of executable tests, an MBT tool generates sets of test cases that are readable by a computer. Later, these computer readable test cases can be automatically deployed. For instance, these test cases can be set of (Python) classes embody incorporate the logic of generated tests.

Offline generation of manually deployed tests refers to generating test cases that are readable by a human and can be later deployed manually. For example, it can be a PDF, Portable Document Format, document in a human language (e.g. Swedish document that explains the generated test steps).

In contrast, online testing directs attention to connecting the MBT tool to the SUT directly and tests dynamically.

1.1.2 Algorithms for Test Case Generation

The most important advantage of Model Based Testing is related to how well the automation can be implemented. Therefore, the models that are formal well-defined functional interpretations or machine readable models can in principle originate test case automatically. It is common to translate the models to *State Transition Systems* (STS) or *Finite State Automata* (FSA) [10]. These STS or FSA show the feasible configuration of SUT. Thus, to generate a test case STS/FSA is ought to find an executable path. An arbitrary feasible execution path works as a test case. This technique is only possible if the model is *Deterministic Finite Automaton*, FDA, or if it is reducible to a FDA [11].

Concerning the complexity of the SUT and the respective model, the amount of paths can be huge, because of the large number of possible configurations of the SUT. To find proper test cases (i.e. paths that point out specific requirement) the search method should be directed. Several test case selection methods have been used.

1.1.2.1 Theorem Proving

Theorem Proving [12] has been basically applied for automated proving of logical expressions. In MBT approaches, the SUT can be modeled by a series of logical predicates to declare the systems

functions and behavior. In order to choose test cases the model is divided into equivalence classes over the sound interpretation of the set of the logical formulas giving an account of the SUT. Every class is standing for a specific system function or behavior and consequently can play a role of the test case.

The trivial dividing is performed by the disjunctive normal form approach. The logical formulas depicting the system's behavior are converted into the disjunctive normal form. The classification three-method creates a more complicated hierarchical dividing. Moreover, dividing heuristics are applied to support dividing algorithms (e.g. heuristics founded on boundary value analysis.)

1.1.2.2 Constraint Logic Programming

Constraint Programming can be applied to choose a test case to satisfy a particular constraint by solving a collection of constraints over a collection of variables. The SUT is depicted by means of constraints. Solving the collection of constraints can be performed by *Boolean logic systems* and solvers such as *SAT Solver*, or by *Numerical Analysis* such as *Gaussian Elimination*. A solution uncovered by solving the collection of constraints expressions can be assumed to be a test case for the respective SUT [13], [14].

1.1.2.3 Model Checking

Basically, *Model Checking* [15] was created as a method to assure whether an attribute of a specification is acceptable in the model. We develop a model of the SUT and a specific attribute that we aim to examine in the model checker. As far as the attribute is under test to get proved, if this attribute is acceptable in the model the model checker identifies instances and contradictions. An instance can be a path where the attribute is satisfied, whereas a contradiction is a path in the execution of the model where the attribute failed. This particular path can be reused as a test case several times.

1.1.2.4 Symbolic Execution

Symbolic Execution is mostly applied in MBT structures [16]. It can be used to find execution traces in an abstract model. Generally, the program run is simulated by applying symbols for variables instead of real values and operations instead of real functions. Then, the program can be run as a test case. Therefore, the symbols should be instantiated by giving values to the symbols.

1.1.2.5 Event Flow Model

Event Flow Models [17] have recently received much attention for testing *GUI*, *Graphical User Interface*, software, and shows events and interactions among the events. Keeping the structure of *Control Flow Model* to show the whole feasible execution traces in a program, plus the structure of *Data Flow Model* to projects the whole feasible declaration and functions of memory location, the event flow model shows the whole feasible series of events that can be run on the GUI. More precisely, an arbitrary GUI is disassembled into a modal dialogs hierarchy. This hierarchy is projected as a tree in which every modal dialog is shown as an event flow graph that represents the whole feasible event execution path in the dialogs and individual nodes. Since the event flow model has no ties to a special aspect of testing GUI, it is possible to apply it to run several testing tasks by declaring specific MBT techniques named as Event Spaced Exploration Strategies, ESES [18]. These ESES utilize event flow model in a number of ways to provide a GUI testing procedure, namely by test case generation, model checking, and creation of a test oracle.

1.2 MBT Automation

MBT automation is to using assistant and automated system to manage and administrate the running of MBT; to check whether real behavior and predicted behavior are compatible or not. It should be initiated by setting up MBT preconditions, MBT controls and MBT oracles. MBT automation often includes automating hand driven procedures. Recently, CASE tools that assist developer to create GUI applications quickly, have had intense progress. This has put the tester under more pressure, which they have already known as the bottleneck of software projects. Testers usually are requested to test more in less time. MBT automation is aimed to make this feasible, because hand script testing is highly time consuming. Furthermore, whenever new versions are released, the new options should be manually tested again. Contrary, we already have CASE tools that assist MB testers in the automation of graphical user interface which decrease the cost and test time, besides supporting the performance. Moreover, with these CASE tools we can exploit the payback and record options that let users record user activity interactively and replay it back as many as desired, comparing real output to those predicted. It should be noticed that confidence in these options raise serious maintainability and reliability obstacles.

A growing trend in model driven engineering is to use MBT frameworks that enable the model to manage the unit tests to decide whether different parts of the code are tested scenarios. MB test cases express tests that need to be executed to assert that the program behaves as required. All of these actions can be automated [19].

From the other point of view, MBT automation includes another concept called *Partial MBT Automation* (i.e. automating parts contrary to automating the whole MBT process) e.g. in the case that an oracle cannot rationally be produced. This would be very troublesome to maintain, then an MBT tool expert can instead produce MBT tools to assist the tester staff to make their tasks more efficiently. MBT tools can assist to automate actions such as *Problem Detection*, *MB Test Data Creation, Defect Logging*, *Product Installation*, *Graphical User Interface Interaction*, etc., in the absence of unavoidably automating MB tests in an end to end mode. MBT automation is costly, but it is predicted that MBT automation will be cost-effective in the future, particularly in regression MBT.

1.3 Thesis Topics

In this thesis we will inspect and analyze applied CASE tools in MBT. Every MBT supported-tool has many aspects to investigate, but what we consider to be interesting to study is divided in two parts. The first interesting topic is documenting the state-of-practice of some quality factors of these MBT tools. A list of quality factors is given below

- Credibility of Applied methods
- Efficiency (Resource consumption: CPU, Tester)
- Extensibility (Accepting another type of model or language or size of application)
- Maintainability
- Platform Independency
- Quality of Results (Accuracy and Precision)
- Portability of Model
- Performance (Response Time, Amount of Reported bugs)
- Usability issues (Support, UI, Navigation logic...)

The second topic which is suitable for academic studies is related to the algorithm for generating test cases. There are many types of these applied algorithms and we have noticed that the companies used extremely different versions comparing each other. It would be interesting to compare the applied heuristics of the product with the rich academic research in algorithms with regard to:

- Efficiency,
- Coverage and completeness,
- Precision and accuracy,
- Extendibility.

1.4 Research Methods

In order to satisfy the goals, we started to deeply go through the MBT theory with the academic approach, since the concept of MBT is relatively new. Meanwhile, we enjoyed from the opinion of

many professors from Linköping University and Helsinki University of Technology. It takes months to cover adequate amount of academic contributions, and get prepared for investigating an industrial case study.

To get narrow to the industrial case study, we spent weeks at the tool vendor, observed activities, followed up the whole phases of MBT from the very beginning of determining requirements to the last oracle part. We monitored how they faced the project, problems and constraints and where they could overcome them and where they could not. We exactly followed their work and identified their positive and negative points. Last but not least, we put their background theories under examination and the comparison with rich academy theories.

By following the mentioned approach we had some trade off:

Pros:

- Preparing very good background both for readers and for the rest of the thesis.
- Touching the tool vendors directly causes erasing any vagueness.

Cons:

- The process was time consuming and had some costs.
- It is too technical and probably most useful for experts in testing and theory.

1.5 Contribution

The contributions of this thesis are follows:

- Survey on current state of Model Based Testing. We will first cover the theoretical aspects of MBT completely.
- Survey of MBT supported-tools. Tools for majority of leading companies are given account by details.
- Comprehensive investigation on Comformiq Qtronic as an industrial case-study. Diving into the chosen tool to discover the distance between academy and industry.

The thesis has practical and theoretical results, suiting for both researchers and practitioners. The tool is not based on our framework.

2. Theoretical Framework

The objective of this chapter is to present a method for test generation by using a graphical notation that gives an account of behaviors' of a SUT.

Applying a model to represent behaviors' of a system has been proven and it provides noticeable benefits to test teams. Models can be applied in variety of fields in the software development, such as: Generating the program code, analyzing the reliability, improving the quality of specification, and the most concerned in this thesis is Test Generation [20].

2.1 An Argument about Modeling

Models are mainly exploited to understand, designate, and develop software in many ways. From nanotechnology to research in constructing the modern fighter plane, models are applied to assist, realize and prepare a reusable framework for system development. In the software engineering

development, models are now approved as a part of modern *Component Based*, CB, analysis and a design approach for all main CB methodologies [11].
Modeling is an appropriate method to receive information about a system and then reusing this information when the system enlarges. For test team, this knowledge is extremely valuable; how much portion of testers' time should be taken by attempting to realize what is the SUT doing? Once this knowledge is realized, then how it is conserved for the next testers, engineer, or even next versions? The best case is when this is documented in the test specification. However this is often forgotten ant a test script or could easily be lost. By creating a model of a SUT that declare the system expected behavior for particular inputs, a test team has the mechanism for a analyzing the SUT. Therefore, scenarios are expressed in a series of actions in the SUT (as it described in the model), with the appropriate reflex from the SUT [22]. Moreover, test coverage parameters are clearly described and test plans are designed to check the functions and behaviors of the system under test. It should be highlighted that the significant advantage of MBT is in reuse aspect; none of these works can be saved and apply in the next round of testing. The next test series continue wherever the previous one is left. In the case that new featured added to the SUT, they can be incrementally summed up with the model. Furthermore, if higher quality would be expected from the SUT, the model and test cases can be enhanced by explaining more details of the states and their behaviors. Finally, new members can catch up the senior engineers by reviewing the models of SUT [7].

2.2 Specification

In this thesis we supposed that the SUT is at the integration or system test level of the software development process; as the arrows show in the figure 3. The inferred test objective is to assure that the system goes on the right track of its requirements from the external point of view. Thus, the concentration is not on the implementation, but on how the testers would evaluate testing. The tests will evaluate the general conformance of the SUT to the specification instead of code coverage. These black box tests are named as *acceptance tests* [5].

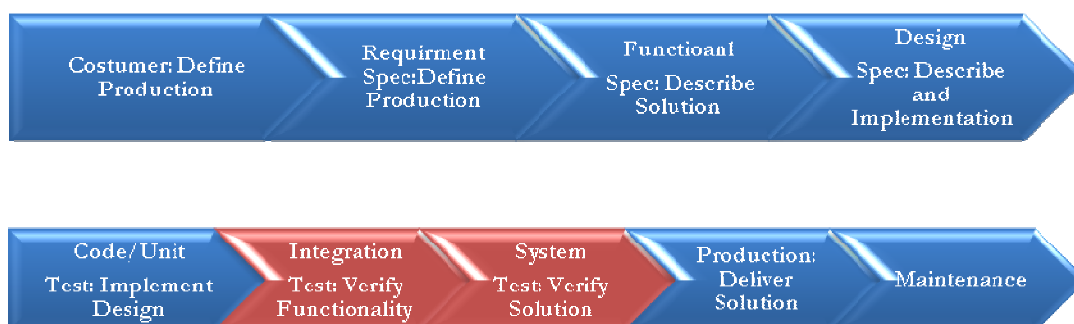
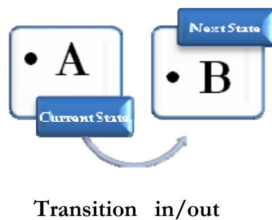


Figure 3: software development process

The first problem in the way of providing tests is to decide on the test target. Although, this issue may seem trivial, it is almost the primary stage that mistakes take place. A description of the system or software to be tested is essential. The shape of the description is widely different from a collection of call flow diagrams for a *Voice Mail System*, VO Mail, to the *Graphical User Interface*, GUI, of the *E-Bank*. A clarified set of characteristic and/or behaviors of a system are required to declare the boundary of the development and test work space. The classical ways of defining the valid system behavior is with natural language prose in the style of *Requirement Specification* or *Functional Specification* [25]. Even though the written specification is a good means to describe the SUT, it rarely mentions all possible scenarios and every detail function. This imperfect specification keeps the tester way up to deliver time so that the whole context of the characteristic is known. When the entire context is understood, tests can be created that will confirm remaining scenarios. The secondary obstacle with textual description is that they are vague i.e. in the case that wrong integer is fed to the system, proper decision should be made [9].The term “proper decision” is never declared; instead, it is left to the readers’ inference. For instance, the programmer might assume that the system should let the user to re-enter the digit, rather than the tester assumption that could be aborting the action. Hence, the code and the test would fail and both programmer and tester waste their time resolving a matter that could have been solved before they even began.

Using a model at this degree of development process can dramatically decrease the vagueness and therefore the errors. The truth is that modeling does not require lots of time to learn. Currently, it is merged in the academy and industry. Testers usually provide models. The remained issue is whether the models are in an invariable shape or not. The model may only remain or be for a short period and live on napkin or in the engineer’s fiction. In order to program a test script or test plan, a tester has to learn the preliminary steps to use the system. Indeed, modeling at the behavioral stage is very comparable to draw a flowchart; the main transactions in the use of the system are declared in a graphical notation. The order of transactions that could take place throughout the use of system is declared. In addition the actions that “could” happen imply that there may be more than one possible way from particular point in the procedure. Majority of modeling methods supports the notion that there could be multiple “next” actions [6]. Many methodologies are founded on the idea of *State Machine* (shown in figure 4), that the transactions shown by arrows in the graph correspond to the actions, as long as there are lots of graphical format to figure the states. Some modeling methods encourage hierarchical models which any state can be substituted by a “call” to another model where the behavior of the state defined. Moreover, hierarchical models let the complicated behavior to be taken part into less sophisticated level models. Supplemental modeling options include technical conditionals, expressions, or predicates to create the transactions i.e. arrows that rely on variables or current context. Many other textual styles and formats have been used in both academy and industry such as: SDL, and Z both popular in communication sector [16].



- Every **Transitions** includes an **INPUT EVENT** and a **NEXT STATE**
- A **Transition** can also declare **OUTPUTS** and **ACTIONS**

Figure 4: Finite State Machine representation

Developing graphical model from a specification, even if performed late in the process is a valuable source since [16]:

1. Finding errors in the SUT (many of them made visible if put them on the model)
2. Quickly declaring the base to use scenarios of the SUT
3. Conserving this resource for next versions
4. Model Developing can be done in incremental steps.

2.3 MBT Fundamental Tasks

2.3.1 Understanding the SUT

As it mentioned above, providing a mental shape of the SUT's functions are needed to create models. This is not a simple job, since majority of SUTs currently have complicated user interfaces and complex behavior. Furthermore, applications are developed within large OS among a disorder of other sophisticated systems.
To develop a realization of a system, testers have to acquire knowledge about both the system and its environment. By requiring the *Exploratory Techniques* ([Kaner et al. 1999]) and studying available documents, MB testers can collect adequate amount of information to create sufficient models. The following are some guidelines to the tasks that may be carried out:

Decide on the part/characteristics that require being tested based on test purposes.

No model is perfect for entirely describing complicated or large applications. Deciding on what to model for testing is a preliminary step in staying in the MBT track.

Start surveying target region in the SUT.

If development has already begun, obtaining and surveying the most recently developed components with the plan of learning about behavior and what possibly can cause errors is a worthy practice toward building a mental model of the SUT.

Collect related and practical documents.

Similar to most testers, MB testers require discovering as much as possible about the system. Exploring requirements, use cases, specifications, relevant design documents, user manuals, and

every document that is available are essential for clarifying vagueness about what the system is going to do and how it does it.

Set up communication among requirements, design, and development teams if possible.

Debating about issues with other teams on the project can save a lot of effort and time, especially when it comes to selecting and developing a model. There are organizations that developing a

variety of models throughout requirements and design. Why not create a model from the very first steps that can be reused or imported for testing purposes, hence saving important time and other means? Additionally, much vagueness in natural language and/or formal specifications can be better determined by direct communication rather than reading bunch of reports and documents.

Recognize the target users of the SUT.

Every entity that either provides or employs system data, or has some effects on the system in some discipline has to be considered. Observe user interfaces; mouse and keyboard input; the operating system kernel, other system calls, file, and network files, databases and other data stores; programmable interfaces that are either proposed or hired by the system. At the beginning, this may seem complicated because many testers and developers are not comfortable with the concept of system having users that are neither human nor programmable interfaces. This recognition is the elementary step to study events and sequences thereof, which would sum up to testers' skill to detect unexpected test outcomes. Eventually, we have to mark those users whose reflex needs to be simulated based on the testing purpose.

List the inputs and outputs of every user.

In some conditions, this may seems like a crushing work, all users noticed, and it is boring to carry out manually. However, separating the work accordingly with user, component, or characteristic, would dramatically decrease the tediousness. Moreover, there are industrial available CASE tools that lessen much of the needed effort by automatically discovering user controls in a GUI. At this point, since automation is often deliberated in MBT, the testers have to start inspecting origins of simulating inputs and detecting output.

Study the domains of every input.

For generating applicable tests in later steps real and significant values for inputs require to be created. A test of illegal, boundary, and expected/normal values for every input needs to be executed. If the input is a function call, then alike analysis is needed for the return value and each of the arguments. Afterwards, intelligent abstractions of inputs can be created to simplify the modeling procedure. Inputs that can be simulated in the same way may sometimes be abstracted as one. It may also be simpler to perform in the reverse: calls of the same function with non-similar arguments may be assumed as different inputs.

Document input applicability information.

To generate useful tests, the model has to contain information about the context that rules whether an input can be used by the user. For instance, a button in a specific window cannot be pushed by the human user, if that window is not open and active. Plus, testers need to remind changes in these contexts triggered by one of these inputs.

Document the situations under which particular reaction can occur.

A reaction of the SUT is an output to one its users or a change in its interior data that affects its behavior at some point in the future. The situations which inputs led to certain reaction need to be investigated. This not only assists testers to evaluate test results, but also design tests that deliberately originate particular reaction.

Study the order of inputs that need to be modeled.

Studying the sequence of inputs leads straight forward to *model building* is the situation where most of the misunderstandings about the SUT are found. The following questions need to be responded; Are all inputs useful all the time? Under what conditions does the system anticipate or accept particular input? In what order is the system needed to handle relevant inputs? What are the circumstances for input sequences to provide certain outputs?

Realize the structure and meaning of external data stores.

Finding the structure and meaning of external data is specifically significant when the system stores information in large files or relational databases. Understanding what the data looks like and its semantics would permit weak and risky regions to be revealed for analysis. This can help to create models that generate tests to originate external data to be corrupted or tests that trigger failures in the SUT with inconvenient cluster of data and input values.

***Realize interior data interactions and computation*.**

As with the last activity, this action sums up to the modeler's realizing of the SUT, hence the model's capabilities of generating bug-revealing test data. Interior data flow among several parts is noticeable to creating high-level models of the system. Interior computations that are basically error prone arithmetic, like division or high-precision floating point operations, are generally fertile ground for errors.

Maintain one living document: the model.

Opinions are different on this, but unless needed by organizational rules, there is little reason to provide a document of all related information. Maintaining a collection of pointers to all documents that include the required documentation is adequate more often than not. Some tasks like studying input applicability and input sequences may not be standard in earlier steps of the engineering

procedures while providing essential information to the modeling process; this is approximately worth documenting. Also, documenting obstacles encountered throughout modeling and the rationale behind modeling decisions is also recommended. In the absence of any documentation, it may be worthwhile to also document all exploratory findings. Last but not least, it is quite rational to annotate the model with remarks and observations particularly if there is a lack of appropriate documentation and if the modeling tools allow it.

2.3.2 Test Case Generation

Since choosing and building a behavioral model is system designing procedure not a testing procedure, at this point it has assumed that after understanding a SUT, a developing team chooses the fittest model type for their system requirements and builds it. The difficulty of generating tests cases from a behavioral model relies on the substantial characteristics of the model. Models that are vital for testing often have elements that make test generation effortless and, regularly, automatable. For some models, all that is needed is to examine combinations of conditions expressed in the model, demand basic knowledge of combinatorics [7]. In the case of FSM, it is as easy as developing an algorithm that randomly (or deliberately) traverses along the machine through nodes and edges. [2, 18]. The order of edge names trough the created tracks are tests case e.g. In the FSM in figure 5, the order of edges from start node meet the minimum requirements of being a test case of the symbolized system.

There is a diversity of constraints on what comprise a path to satisfy the requirements for test cases. Examples contain having the path start and end in the beginning state, limiting the number of loops or cycles in a path, and limiting the states that a path can meet.

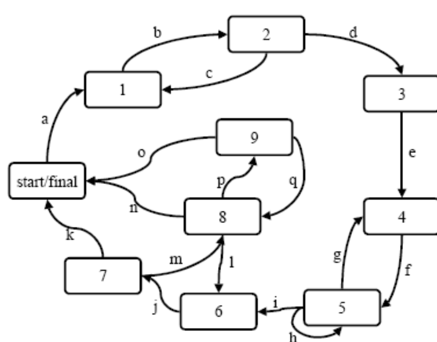


Figure 5.A: A Finite State Machine

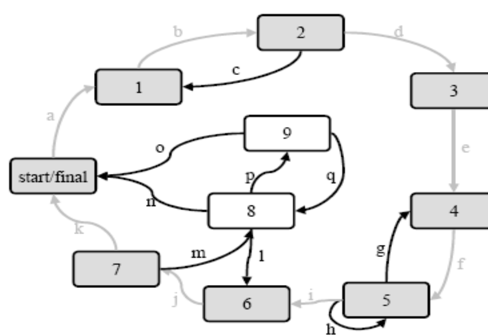


Figure 5.B: Another Test Case (Path) in the State

2.3.3 Generating Expected Outputs

Testing includes execution of a SUT using some error-revealing input data and evaluating the output to decide success or failure. **A significant hypothesis for this testing is that there is some mechanism, a test "*oracle*", that will determine whether the output of a test is correct or not** [14]. As clarified in the introduction, **expected outputs should be produced prior to execution of tests**. An **oracle is the criterion employed to verify the correctness of the results**. For instance, the behavior of a competing: application possibly the basis for evaluating the properness of the application under tests "It should perform what application B performs". Another example would be applying a latest version of the product in which the unit/characteristic under test did not endure significant change i.e. "We should get the same output now that we got with this in version X.". Creation, adoption and function of the oracle is almost depends on [19]:

- Size and complexity of the SUT
- The level of automation in the testing procedure. The larger the size and/or complexity of the system, the more extensive is the need for automation. However, automation itself makes writing/using an oracle more troublesome.

The oracle requires to be provided in such a way that it can "couple" expected behavior with corresponding tests so that success or error can be decided automatically for the millions of test cases typically generated for a complicated system [23]. Additionally, the oracle requires being flexible enough to simply modify to the dynamics of test generation.

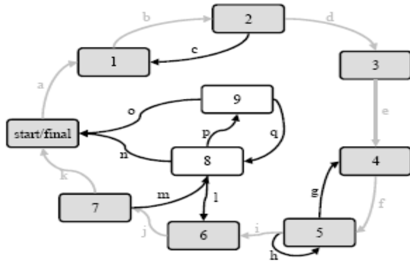
Searching/creating an oracle can be an obstacle in model based testing. Tests cases are generated automatically and comprehensively. Moreover, test suites do not remain static. Therefore, computing expected outputs by hand is usually impossible. Some work has discovered the automatic generation of oracles. In lack of a good oracle, one may need to arrange credible verifications. Tests cases may be believed to have succeeded if their outputs are in specific ranges or they succeed particular consistency verification. If the system is equipped to recognize its state, expected test outputs can contain the system state. In many cases, this expected output would be produced automatically from the model connected test inputs.

In actuality, this is almost done by comparing the output, either automatically or manually, to some pre-computed, apparently correct, output. However, if the system is formally documented it could be to use the specification to decide the pass or defeat of a test execution.

2.3.4 Running the Tests

In spite of test cases can be executed as soon as they are generated, it is typical that test cases are executed after the whole suite that satisfy adequacy requirements is generated. At the start point, test codes are deployed to follow the use of inputs by their corresponding state as it shown in figure 6.

Then, the ATS can be simply converted to an ETS. Alternatively, it is possible that the test case generator produces the ETS straight by annotating the arrows with simulation procedures calls.




```
Procedure a() {

// Simulation code for
input a

}

Procedure b() {

// Simulation code for
input a}
```

```
Procedure Test Script() {

// Code to drive software

// into start state

a();

b();

d();
```

Figure 6.A: Test case path in FSM

Figure 6.B: Simulation code

Figure 6.C: Test Script for the test path

During writing the automation script, hiring good engineering practices is demanded. Scripts are restricted to interact with each other and evolve as the system evolves. Scripts can be applied as long as the system is under testing, so it beneficial to investigate some time in writing well-structured and efficient ones. With MBT, the number of simulation process is in the order of the number of inputs, so they are usually not too time-consuming for developers.

2.3.5 Comparing expected output with real output

It is useless to have the opportunity to generate and execute millions of test cases if you do not have a possibility to appraise the outputs and take further action based upon the outputs. Inspection trough test output is approximately the hardest task in testing. Inspector should decide whether the SUT created the valid results by having the set of test inputs which fed to it. Technically, this indicates checking the real results, checking the saved data and creating the resource perquisite were satisfied [24].

MBT does not comfort this circumstance. Results and internal variables must still be verified against the system requirements for correctness. However, model based testing does sum up a new dimension that is very effective in practice: the verification of state. States are abstractions of interior data and as such can often be more simply checked. A case in point, the model automatically lists every node modification that happens in the SUT as inputs are being used. Consequently, the model behaves as a highly accurate specification, tells the test team which inputs have to be ready and the measures of every ADT that includes the current conditions of the SUT.

Much information can be concluded from MBT test results that some of them have been listed below.

- A tester behavior model which extra test case can be generated
- Test cases, contain expected behaviors for the test cases

- Degree of test coverage reached by the generated test cases
- Test outputs from which the reliability of the SUT can be predicted

In Addition model based testing eases management decision making in senses of:

- Testing termination condition and release a new version
- Model Correcting
- Extra test case generation

2.4 Key Observations

2.4.1 State Explosion

There is a concern of state explosion in many scientific papers about creating, maintaining, reviewing, verifying, and testing using models. In fact this obstacle can never be really appreciated without an example. Basically, there are two methods to take care of state explosion if there is no possibility to be faced directly, *abstraction* and *exclusion*. *Abstraction* for model based testing includes the unification of compound data into a less complicated system e.g. a webpage that needs to fill in the multiple spaces of data and continued by the **approve** link , may be reformulate as two simple action: "fill in correct information" and "fill in incorrect information." Consider that the abstraction catches the end output of the data that is entered so that a transition to the valid state can be produced. Therefore, when the data is incorrect, the model shows sufficient information to transition to the invalid state. However, the functionality of abstraction always has the effect of losing some information [25]. In this case, we miss the information about precisely which space field was entered in erroneously. We easily know that as a suite, the data is invalid. Clearly, abstraction can be abused. But when used intelligently, it can hold the significant information and ignore unnecessary or irrelevant information without lack of efficiency [25].

Exclusion simply refers to relaxing data without concerning to decrease its complexity. It is usually done when taking a part of a group of system functions into different models. Every model would include specific data to feed in and some data to list out. The feed-in data that listed out by a model builder would have to be checked in several methods, clearly. State explosion depicts a critical challenge to the MB tester but by no means does it eliminate the use of FSM as a model, for there are no silver bullets in testing [25].

2.4.2 Test Coverage Parameter

For FSM, coverage is defined regarding to feed-in data, nodes, and edges. Indeed, small models will let the designer become happy situation of full state and transition coverage. More complex models

are troublesome and until more research is carried out, MB testers should choose from the large options of graph coverage algorithms available [23].

2.4.3 MBT Advantage

Employing MBT causes lots of benefits such as [3, 9, 19, 21, 22, 23, and 24]

- Better quality and reliability, less cost, and saving time
- Keeping a user behavior model constantly
- Establish communication among developers and testers in conjunction with building the model
- Early revealing of vagueness in specification and design as far as building the model
- Potential to automatically generate millions of non-repetitive and practical test cases
- Test harness to automatically execute generated tests
- Joining the MBT artifacts eases the updating of test suites for altered requirements (usually, only the model required to be updated)
- Potential to assess regression test suites (one can know what degree of test coverage they reach)
- Potential to evaluate SUT quality
- Testers felt wonderful profit in having millions of test suites generated automatically, and plus they were glad to reveal that those tests could be updated by easy modifications on the current model.
- Programmers were not interested about test case generation and maintenance, but were happy that their code was being suitably tested.
- Decision makers had the most difficult period observing the advantages of MBT. They could observe that models were generating more test cases than previous, but it was more difficult to observe if the tests were performing a better job testing SUT. Test generation upsets current metrics, such as test case number, that directors had before hand relied on.

The complexity of modern systems is growing, and classical, hand-written test suites have become awkward and fragile. MBT can produce a powerful increase in testing abilities, but modeling technology should be mixed into everyday testing. Small-scale pilot system, readily available CASE and tester education have made the transfer to test generation simpler at large companies such an Ericsson AB.

Modern systems are flooding into the classical methods of testing.

As the project life-cycles lessen and systems become more complex, common testing approach are not revealing the significant bugs before release. Organizations are giving notice to new technologies that can assist them reaching the quality they required.

Testers are becoming more specialized.

The era of the non-technical tester are coming to end. Organizations are finding out that they need technically expert testers to manage software testing, and these engineers are more productive if they are engaged early in the phases. Engineers who are comfortable with modern approaches are more welcome to trying test generation.

Test models can apply as executable specification.

Natural language project specifications are too ambiguous to create guidance testing. Almost they are not updated to move forward with the project. Test models indicate how the system should act and can play a role of formal specifications for the aimed function of the SUT. Since the test cases are generated from this model, the model/specification is kept up to date with the SUT.

2.4.4 MBT Obstacles

Testers are almost non-technical.

In majority of organizations, the testers is noticeably less technical than the programming team; most of the time, they are engineers who "failed to meet the bar" for a programming position. The short technical background of this pool limits the amount of test heuristics that can be absorbed by the team.

Testing is viewed as a back-end, ad hoc task.

In most organizations, testers are not engaged until the application has been designed and developed. Classical testers hence have little impact on system design, and are almost incapable to automate testing.

Testers perform in a severe rush time.

Testers are frequently stressed out to uncover bugs and test cases in the less time. They are deterred from providing test case generation systems that possibly is more cost-effective than classical methods in the long execution.

Formal requirements are uncommon.

Advanced businesses, such as communication and avionics, have detailed specifications. Many of software development companies, however, have not many documented specifications; people work from natural language user scenarios that are substantially vague.

Common test metrics do not map simply onto test generation.

Current test metrics such as "Number of test cases generated" lose their relatedness when test cases are generated by an automatic system. Test engineers are almost unable to translate the status of a MBT project if they only depend on classical metrics.

Test generation is not encouraged in the industry testing documents or CASE.

Testers do not study academic documents. At best, they study documents on testing and industry magazines. These means have little to assert on test generation, concentrating instead on test writing for regression tests. General industry CASEs also encourage capture/replay and test scripting, not test automations. Decision makers are conservative to invest in a technique or a CASE that does not have widespread business.

3. Tools for MBT

In this chapter the focus is to gather the brief information of main CASE tools for MBT generation in both the academic and the industry arena. The goal of the chapter is to observe the tools in the context of what is currently available in both commerce and academia.

3.1 What is the MBT CASE Tool?

Generaly the entire software testing process is based on a model, because any test case that has to be provided applies some mental model of the SUT. Recently, the function of explicit graphical notation in software engineering (most remarkably the advantage of UML for OO analysis and development) has extended comprehension. Exploiting of these models for the generating test cases in the software industry is still in the early stage of development, although an important section of the telecommunications, aerospace, and micro-electronics industries have been testing with models for conformation and test case generation for more than a decade [26].

We will declare an **MBT generator** as automated activities which agree on two principal inputs:

- A formal or semi-formal model of the SUT, and
- A series of test generation methods which direct the tool in its generation.

In many CASE tools, the test generation methods are not equipped explicitly by the tool, but are permanently assisted by the structure of the test generation tool. The output of a test generator is a set of test cases, which contain a sequence of simulation of the SUT, and the predicted responses to those simulations, as expected by the model.

The procedure of MBT generation may be fulfilled manually or with ordered tools produced for the explicit intention of creating a test suite for a single application. A good case study is explored in Robinson's articles [55]. The concentration of this report is on *generic* MBT generation CASE tools.

We make a difference between test generators and **MB input generators**, which do not produce the predicted behavior of the SUT. The models approved by an input generator are models for input series acknowledged by the SUT, but they do not model the predicted output of the system. In order to require such tools for the automation of test generation, the user must also create an oracle for the SUT behavior.

We also distinguish between **test automation frameworks** and **test generators**. A test automation framework approves hand-driven, automatically generated, or pre-derived test sets and executes the collection without human control.

Another level of tools which we do not embrace in this chapter is the set of **modeling tools**. Tools in this class contain Rational Rose [27], Objecteering [28], Poseidon [29], Together Control Centre [30], Statemate Magnum [31], AutoFocus [32], SimuLink [33], SCR [34], and Telelogic Tau [35]. These tools filled the role of creation of input for MBT generators, and do not mainly have the capacity to create test cases.

The table below briefly shows [36] the most famous CASE tool in Model Based Testing. In the rest of this chapter we will give a short introduction for some of them.

Tool Name	Organization	Inputs	Outputs
AETG Web Service	Telcordia Technologies Applied Research	Tabular definition of input parameters	Test cases
http://www.argreenhouse.com/demos/			
Abstract State Machine Language (AsmL)	Microsoft	XML and Word	Test generation based on total transition coverage of FSM

http://research.microsoft.com/foundations/AsmL/			
Conformiq Test Generator	Conformiq Software, Limited	UML state diagrams	Test cases in TTCN format, including expected results. Test harness.
http://www.conformiq.com/			
Direct – To – Test (DTT)	Software Prototype Technologies	Models in custom language, cause and effect tables	Test cases including expected outputs and executable test scripts
http://www.softprot.com/			
GOTCHA – TCBeans	IBM Research Laboratory in Haifa	Model in custom language	Test cases, including expected outputs and test translation framework
http://www.haifa.il.ibm.com/projects/verification/gtcb/index.html			
MulSaw	Massachusetts Institute of Technology	Model in Alloy modeling language or in Java Modeling Language (JML), including pre and post conditions.	Test cases based on coverage criterion.
http://mulsaw.lcs.mit.edu/			
Reactis	Reactive Systems, Incorporated	Models in MatLab's Simulink and Stateflow modeling language.	Model simulator; test suite, including expected results
http://www.reactive-systems.com/			
SDL And MSC based Test case Generation (SAMSTAG)	University of Fribourg	SDL system specifications, MSC test purposes	Test cases in TTCN format
http://diuf.unifr.ch/telecom/samstag/ (Site possibly no longer active).			
SpecTest	George Mason University	Models in SCR or UML	Test cases based on coverage criterion.
http://www.isse.gmu.edu/~aynur/rsrch/SpecTest/			

Telelogic Tau TTCN Suite	Telelogic	SDL model	Test cases based on coverage criterion, including expected results.
http://www.telelogic.com/			
Test Generation with Verification (TGV)	IRISA and VERIMAG Laboratories	LOTOS, SDL, or IF specification model	Test cases in TTCN format, including expected results.
http://www.irisa.fr/pampa/VALIDATION/TGV/			
Test Vector Generation System (TVEC)	TVEC Technologies	Behavior model in proprietary language or SCR model.	Abstract test cases and executable test program.
http://www.t-vec.com/			
The Object-oriented Software Testing Environment (TOSTER)	Warsaw University of Technology	UML state diagrams	Generates and runs test cases, generates expected outputs.
http://home.elka.pw.edu.pl/~alasota/			
TorX	University of Twente	LOTOS, PROMELA, or SDL model	Test cases.
http://fmt.cs.utwente.nl/tools/torx/			
Unified Testing and Specification Toolkit (UniTesK)	Institute for System Programming of the Russian Academy of Sciences (ISPRAS)	Model in custom specification languages for Java and C++. Pre and Post conditions.	Test cases, test driver satisfying branch coverage of post conditions.

The following descriptions are summarized from the surveys of current tools in AGEDIS project website [26].

ASML

AsmL is the Abstract State Machine Language [37]. It is an executable specification language established upon the theory of Abstract SM. The common version, AsmL 2, is embedded into Microsoft Word and Microsoft Visual Studio.NET. It applies XML and Word for literate requirements.

There is a test generation tool which runs with this specification language, and Microsoft investigators have given an account on its properties [38]. Their test generation algorithm seems to be absolute transition coverage of the extracted FSM. The tool is not commercially available.

Conformiq Qtronic Test Case Generator

Conformiq Software Ltd. has produced new software that was due for release in autumn 2002. The notes on their documents expressed that their tool will be a true MBT generator based on the definition of this chapter. The tool receives UML state machine diagrams as the graphical notation of the SUT, together with real time properties. The significant output of the generator's analysis is a behavior simulator. This tool will be studied in depth in section 4.

GOTCHA-TCBEANS

GOTCHA-TCBeans is the among forefathers of the AGEDIS test generation and execution tool suite. The GOTCHA-TCBeans tools are completely explained in two articles [39, 40].

The modeling language applied in GOTCHA is an extension of Murphi [41]. The Murphi language is expanded by the supplement test generation methods that permit the specification of arbitrary projections of the state space to be exploited as coverage standard [40].

The test cases contain predicted behaviors, and the test conversion framework is created by TCBeans. The tools have been required within the IBM Corporation

MULSAW

The MulSaw project [43] at MIT combines two tools for the creation of test cases for Java platform programs: the first one (TestEra) accepts input in the Alloy modeling language, and the second (KORAT) approves input in the Java Modeling Language (JML). The JML is like the language used by UniTesK in that it set pre-conditions and post-conditions in a Java-like syntax as comments above a Java method.

The KORAT test cases are generated to cover all occasions of the method preconditions, and provide expected behavior based on the post-conditions. The tools are not available for experimentation.

REACTIS

Reactis [44] is a CASE tool developed by Reactive Systems Inc – a company that focuses on tools for developing embedded systems. It acknowledges models in the SimuLink and StateFlow modeling languages [45].

The model is compiled and a model simulator is created that can be exploited for manual or accidentally test generation. Moreover, Reactis has an automatic test suite generation potential and coverage methods that are connected to syntactic and structural coverage of the model.

SPECTEST

SpecTest [46] is an automatic test case generator from George Mason University. It approves models designed in SCR or UML, and generates test cases based on an option of two coverage criteria, with a next two planned for development. The tools are not yet available for downloading.

TAU TTCN SUITE

Among the Telelogic products is an application for the generation, simulation, and modification of SDL models. The Telelogic Tau TTCN Suite [47] documents depict a capability to generate TTCN test cases from a SDL model. No specific information was given concerning the use of testing methods; however, TTCN test cases definitely embraced expected outputs that originated from the behavior described in the SDL model.

TESTMASTER

The Teradyne Corporation developed an MBT tool called TestMaster (see [48]). This tool had a proprietary graphical language for modeling the SUT. The tool generates test suites by exhaustive crossing of the FSM applied in the requirements. Test Master had no special facility for translating ATS to executable test scripts. The tool is not currently supported or sold.

TGV/CADP

TGV is a test case generator produced at the IRISA laboratories .The input in is LOTOS, SDL, or IF specification model. The output of the test generator is in the TTCN format, and includes the expected outputs of the test. The test generation methods may be in the form of FSM of the test intensions. The tool has been used comprehensively in academic and industrial situations, specifically in the telecommunications industry. The test generator is included in the CADP package and can be downloaded from the website at [49].

TORX/CADP

The TorX is an architecture for test generation and execution from the University of Twente. Within this architecture there is a test generator that receives test goal in a comparable format to

TGV. The modeling language supported by TorX is LOTOS. The TorX test generator is also blended in the CADP package [50].

TOSTER

TOSTER [56] is a test case generation and execution system developed by the Warsaw University of Technology. It unifies technology for mapping the information in UML diagrams to the source code of a system. It also provides and executes test cases based on expected behavior extracted from the UML state diagrams. There seems to be two test generation algorithms, but no explicit testing method.

TVEC

T-VEC Technologies owned a Test Vector Generation System [51] that approved models of the specifications and behavior of a system in a proprietary language called T-VEC Linear Form. The tool also offers a graphical environment for generating Software Cost Reduction models and automatically converts them into the Linear Form. Convertors exist for two other specification languages namely SCR [52] and MATRIXx [53].

The test generator creates test cases founded on domain testing theory. It creates test cases by analyzing the logical predicates in the specification model, and generating test cases that achieve extreme values in the decision path (a kind of branch coverage of the requirements). The test generation methods seem to be implicit in the sense of algorithm which used to create test cases. The test cases generated by T-VEC contain expected results.

The T-VEC application also has an interface for converting the ATS into ETS. The software appears to have been required generally in the aerospace industry. It does not come into view to have any support for UML or other widely used modeling languages.

UNITESK

The UniTesK (Unified Testing and Specification Toolkit) organized by ISPRAS (Institute for System Programming of the Russian Academy of Sciences) is on the edge between an industrial and an academic tool [54].

The model is explained in either J@VA or C@++, which are specification languages planned for Java and C++ code. The requirements are in the shape of pre- and post-conditions, and are developed as comments in the classes and methods to be tested.

Similar to the TVEC tool, the test cases are created by using branch coverage of the requirements of the post-condition, and this test method appears to be implicit in the test creation procedures.

Since UniTesK is adjacent to the source code, the test methods are generated as part of the test creation procedures, and not in a disconnected abstract to concrete conversion level.

UNITESK has been applied by NorTel in testing functions of the kernel of a real time operating system.

4. An Industrial Case: Conformiq Qtronic

In this chapter the CQ has been chosen for deeper exploration as an industrial case study. This chapter is the result of my study on CQ properties and documents, extraordinary supports from my supervisors, and the warm and kind environment in the Company.

4.1 Brief History of Conformiq Co.

Conformiq Software Company has been established in the last years of twenty century, 1998, by Mr Juha Viskari in Finland to produce technical methods for Model Based testing. This has been the most prominent aspire of the company ever since. Moreover, the significance of year 1998 can be more sensible when it has come to mind the advent of Swiftest, and the initial stage in the succession of decent of the company's test case generation technology.

Giving to births as a solely small scale firm, the company started switch into a factual company in 2001. In the early 2002 Conformiq Test Generator, CTG, the incarnation of the tool's next generation technology came out to the market. CTG is an MBT tool established upon the "test model" paradigm and has acquired attention especially in specific domains of mobile device testing.

The influence of the company's presence technology "Qtronic" started immediately when CTG made available in market. The first Conformiq Qtronic, CQ, version eventually felt the daylight at the last months of 2006. CQ is a genuine MBT tool in the sense that having a functional model of

the approved system, CQ is capable to generate a set of tests to cover the model and therefore the system behavior to a desired stage of exhaustiveness. Last but not least, the company sets up its mission to help companies to lessen Software Quality Assurance (SQA) costs with the assistance of model based software testing and validation tools.

CQ is the consequence of more than five years of successive developing and programming. It is established upon advanced discrete computation and theoretical computer science; yet it is a very practical application. The advantages that CQ presents to software programming are real and spreading. It decreases hazards that happen by incidentally lost or imperfect tests. In the rest of this chapter the value proposition of CQ will be introduced.

4.2 Introduction to CQ

4.2.1 CQ Design and validation phases

From the abstract view, software engineering can be observed as comprise of separated phases of *design* and *validation* [16]. Design is concerning to forming commercial requirements and architectural scheme, programming executable code, and providing implementations. On the other hand, validation is regarding to validating what has been designed up to now concerning to other described issues likewise specified requirements.

A case in point, in the classical "V model" [18] there is an early stage of design: procedure starting with commercial requirements and concluding with implementation. This is pursued by a validation as an intermediately stage that starts with unit testing and continues up to post-deployment supervision. In more late procedural models, such as those under the domination of "agile processes", design and validation are more interlaced. Yet, design and validation are almost two basic components of the procedure. The implicit reason is situated in the unaware mind of people: human errors are inevitable, and therefore everything human makes should be over verified to assure its characteristics. This is correct also in the atmosphere of software engineering.

The design-validation phases are a basis attribute of all software procedures. Some of the most excellent determined techniques for *validation* contain testing, inspections and checking, and static review. CQ is a mean for improving the efficiency of test design like wise the entire design-validation phases in capital. However, it is not a mean for source code inspection or static review.

4.2.2 CQ Testing Cost Assessment

Software testing is a wide field of notions and procedures. Currently, it might be the most significant method for software verification. Software testing uses up prominent quantity of recourses (e.g. money and time). The separation of testing expense relies on how testing is managed. Common methods to manage testing contain [18]:

• Hand driven or classical testing

• Record and replay

• Extreme programming.

Hand driven or classical testing [18] refers to a testing expert or even a tester that deals with the SUT individually, and almost pursuing a blueprint which sketched in a human readable language and providing reports of his/her knowledge which obtained from the system as well as any imperfection that already marked. The ruling fees are human resource fees which originated directly from the testing endeavors on day to day basis.

Record and replay [20] is a greatly deployed model for testing application with GUI. At the very steps, a tester deals manually with the SUT via the UI. The mutual action is saved in an appropriate way. In the next stage, the interaction may be answered several times and the results of the system contrasted to what predicted with the best results that concluded either from the pure running or from other arranged data tables. In record and replay the costs are ascribed to the output of the scripts in the primary place, and the maintenance and manipulating of them afterwards when the output or its requirements altered throughout the life cycle beside the verification of those cases where tests are unsuccessful for diagnosis.

Record and replay surpasses in a procedure where advanced versions of the identically software should be tested several times. Record and replay reaches comparative economics of scale over hand driven testing when the amount of regression test execution rises.

Similarly is correct for applying *extreme programming* [7]. This is a common method to manage regression tests for not large scale units; however, it is used also for extended systems. In this method, a testing expert makes and maintains custom software while running, test the other SUT. The first development has a price for extreme programming may be larger than for record and replay — at least a variety of expertise collection is needed — yet in the long execution it possibly is more effective. Generally, an extreme programming plan can create millions of distinctive test inputs to a SUT, and can analyze the output from the SUT in a much more itemized way than an ordinary record and replay technique.

Since testing is finally cross-checking a source code versus the requirements, the whole forms of testing make costs relevant to comprehension and reviewing requirements. In the manual testing background these expenses expose a working hours expended by test team meanwhile the testing process itself. For extreme programming, both test plans as well as review of noted failours acquire costs (automatically marked failures should be entirely reviewed due to the fact that it might be the testing software itself and being just another computer program developed by a programmer could be error prone itself).

To summaries: a rough but adequate way to classify the cost drivers of a testing process is:

1. Determining and reviewing requirements
2. Producing and maintaining test objects (recorded interactions, custom testing software)

3. Running tests (either classically or by running automation tools)
4. Reviewing test outputs
5. Collecting the information and reporting

4.2.3 CQ in Software Process

CQ is an appropriate means for automatic test case generation that is stretched out from abstract models. This implies that CQ formulates test cases for a SUT automatically when it is fed an abstract model of the SUT as an input. The tests are going to run in black box mode, indicating that they assess the SUT established only upon its exterior behavior, not on supervising its interior activities directly (white box mode).
The behavioral model is a depiction of the planned behavior of the SUT on some degree of abstraction. On the other hand, it is correct to see this model as the most perfect reference for developing the SUT, even though generally it would be the high level and easy to understand. In CQ this abstract model can be provided as

1. A Semi-Java program (a special programming language that has been created by the company and has its root from JAVA)
2. A collection of UML state machine by using the Semi-Java PL as an action language
3. CQ λ, a dialect of LISP (a special programming language that has been created by the company and has its root from LISP)

Additionally, behavioral models can be assumed as functional requirements. They give an account of the aimed exterior functional characteristics of the SUT in a sense of how the SUT should behave from user point of view. Behavioral models do not require presenting the factual implementation, since they express the intentional apparently observable attributes.
CQ selects and via choices performs test cases automatically established upon the behavioral model, and estimates anticipated answers from the SUT automatically. By using CQ, there is no demand to make hand driven tests. Planning the tests, optional running and reviewing are done automatically. These advantages clearly lessen expenses and risks. However, behind this degree of clear advantages, CQ leads in a spreading modification to the software process: it joins design with verification!

Contrary, traditional testing engages human-based translation of requirements into expected results and final judgment. This activity is performed either by a classic tester, a test planner, or test engineer who scripts the testing environment. Commonly, an extreme programming is just a new manifestation of the requirements for the SUT, yet this time in the shape of an executable format that verifies the SUT against the requirements in particular nominated conditions (test cases). These test scripts contribute to keep two subjects in parallel: the requirements and the software testing.

This origin of costs and risks, time, can be diminished by applying such a MBT tool like CQ, since the tool creates tests straight forwardly from the behavioral model. This carries in an extra advantage: test cases do not require to be kept and the quality of the requirement specification

growth intensely. Eventually, when the tests provided by the tool from a behavioral model pass, says that the SUT and the requirements are coherent. This enlarges the value of the behavioral model as a technical specification for the SUT.

4.2.4 CQ Advantages

The most significant advantage of employing CQ is a multiplied product quality that is reached by exploiting the behavioral model as the best resource for implementation of the SUT. Dissimilar to other test tools, test cases can be automatically provided from the behavioral model.

While online testing mode is intended, CQ provided a mass of different test cases from the fed behavioral model. It can utilize the model to directly check a running SUT or to create test scripts that can be freely performed subsequently. Produced test scripts can be saved in a version handling system permitting tests to be dispatched to colleagues or to be performed independently.
Moreover, by online testing, also automatic test case generation from SUT models decreases risks and expense: It removes the risk of deficient test cases and decrease costs by clipping the volume of classical test maintenance activities.

One of the most apparent advantages of employing CQ is that automatic testing established upon behavioral models conserve effort as there is no demand to maintain varieties of test cases and test plan.

Since CQ generates test cases by appraising the behavioral model; it is capable to derive test cases that could be otherwise ignored. Furthermore, it decreases the risk of imperfect tests as the tests are derived directly from the behavioral models. For particular and important tests, tester can design different use-case tests expressing definite specific behavior that has to be clearly tested; hence we create more accurate model and more test cases.

Employing functional models as tools for testing has a strong effect on the quality of functional models. It also serves as a documentation of the SUT. Whenever an error is uncovered between the models, the implementation can be upgraded. This implies that the SUT documentation is usually up to date and matches with the SUT.

Since the behavioral model has such a significant role, CQ has to propose a *model debugging* option — while the functional model is being developed, CQ can be used to decide that there are no execution paths that would cause interior calculation errors (e.g. division by zero). If CQ encounters such an execution, it supplies a counter-example with the analogous execution track and data values empowering the user to correct the model.

4.2.5 Anatomy of CQ

CQ is a tool for both online and offline testing of a SUT and offline test case generation. CQ generates and runs test cases originated form the behavioral models. It is not built-in; however, it is

ameans for generating such a functional models. The cause behind this is that CQ upholds various kinds of models, and several formats are provided and adjusted by applying different tools:

- Java/UML models can be provided by employing editors and CQ Modeler, distinct application that is bundled with CQ.
- CQ models can be provided by applying editors or λ programming.
- CQ Ecosystem Packages (QEPs) are produced by applying specific application to administer ecosystem licensing.

To begin testing with CQ one requires first a *behavioral model*. Relying on the selection between online testing or offline script generation, it needs in addition either a *system adapter* (regarding online testing) or *scripting back-end* (regarding offline test generation).

System Adapter: CQ has to be linked to the SUT with a tight connection. This infers that CQ prefers to communicate directly with the SUT that is going to be tested. The link between CQ and the SUT is coordinated by an adapter. Simply, an adapter is a plug-in that joins to CQ applying a well defined API and joins with the SUT in any appropriate method.

In a more modern setting, various adapters can be employed in parallel: they can be distributed over TCP/IP, SIP networks, and several data filters and manipulators can be supplemented before the adapters. These potentials are generated by the standard components provided with CQ. The only custom components that must be created are the basic adapters.

Logging Back-End: Logging back-ends are employed to create logs and reports from the tests that CQ would perform. It is also a plug-in that links to CQ using a well-defined API. Various logging back-ends can be applied in parallel: they can be distributed over TCP/IP, SIP networks, and several data filters and manipulators can be supplemented in front of the back-ends.

This potential is supplied by the standard components created with CQ. The only custom components have to be made are the basic logging back-ends.

Scripting Back-End: There are software processes where it is advantageous to create distinctive test code that can be saved in version handling systems, possibly sent around, and performed independently and subsequently. To satisfy this requirement, CQ produces the methods for creating test scripts from SUT models where test cases are generated automatically from a behavioral model and can be run against a real system. As SUT models express the desired behavioral of the system, automatic test case creation from SUT models lessen risks and expenses. Basically, a scripting back-end is a plug-in that is linked to CQ requiring a well-defined API.

CQ is delivered with two scripting back-ends. First one derives a traceable HTML file and the second one produces TTCN-3 test script which empowers occupation of MBT in a TTCN-3 sphere. With TTCN-3 script back-end, TTCN-3 test cases can be generated automatically from a behavioral model and be run against a real SUT.

Several scripting back-ends can be used in parallel: they can be distributed over TCP/IP, SIP networks, and various data filters and manipulators can be supplemented before the back-ends. These options are created by standard components provided with CQ. The only custom components that must be created are the basic scripting back-ends.

4.2.6 Design of CQ

The implementation of CQ has been founded on two main purposes: *industrial applicability* and *openness*. This means that designers have strived to produce an application that can be used in solutions by ordinary software engineers in companies, and that can be merged easily into heterogeneous application development tool chains.
The most apparent outcome is that CQ permits the engineers to employ modeling in many patterns that are determined to make obstacles for model verification algorithms such as class coupling. Particularly, CQ upholds limitless data types, complete dynamic data with garbage collector, classes linked to static and dynamic polymorphism, concurrency, and timings. Moreover, the tester is not expected to make abstractions or clipping the model, because the design concept is that CQ should ease the tester of such technical details. However, approved models can have freely complicated and vast state spaces. Therefore, soon we can have a model that is very troublesome for CQ to handle. The solution to this obstacle is that CQ does not force to solve it, since to provide the tool's raison confidence. It would be adequate that there is *fairly* a context where CQ can make solid value.
Another result is that variety of the accredited algorithms for model verification and test automation that suppose FS specifications, or even exclusively finitely dividing specifications, are good means in the tester's hand. It should not be claimed that study in MBT from FSM would be not practical generally, particularly as there are contributions that confirm otherwise.

4.2.7 Modeling in CQ

The tester who uses CQ must manage is the model, and it is very prominent that the model can be given in account of a forceful language that is simple to adopt. Yet the obstacle is creating an industrial scale specification and modeling language is a challenging.
In the public release of CQ, the functional models are unions of UML State Machine Diagram and pieces of an OOPL that is generally a superset of Java or C#. It is a half-graphical notation and named Qtronic Modeling Language. The function of UML state machine diagram is voluntary so that models can be expressed in pure programming language also.

QML is an extended version of Java by summing up value type records (Already included in C#), static polymorphism, discretionary type inference, and free-form macros. It should be emphasized that that QML does not contain the standard libraries that carry in Java or C#.

A QML program should indicate a platform independent system, (i.e. a system with one or more message transferring interfaces that are ready to see from the environment). These interfaces fit to the test surface of the real SUT. For example, if a QML program begins by transmitting out a

message A, then a complying SUT should also transmit out the message A when it begins. Hence, in a very substantial style, a QML program is an abstract developed schema of the SUT it describes. Indeed, a suitable method to test CQ itself is to execute tests generated from a model against a pretended run of the very identical model. Figure 3 presents the association between a model and a related IUT (implementation under test). The test harness spans the more abstract level model with the real working IUT.

4.2.8 CQ Main Functionality

CQ suggests two main supplementary functions for deploying the generated test case: *online testing* and *offline test generation*. Online testing indicates that CQ is linked directly with the SUT by means of a DLL plug-in interface. In this style, the choice and running the test stages and the checking of the SUT's behavior all occur at the same time. Contrary, *offline test generation* separates test design from the test run. In the offline style, CQ generates a set of test cases that are sent out by means of an open plug-in and that can be ready later, free of the CQ application.

CQ supports the testing of SUT against *non-deterministic automata*, i.e. again models that let for various visible actions *even versus a deterministic testing policy*—yet only in the online style. However, upholding nondeterministic automata is something troublesome to develop.
Currently, the offline testing supposes a deterministic automaton. The main reason behind it is that the test cases related to nondeterministic automata have a similar appearance to trees as at the test creation time the selections that the SUT will make are not yet determined. The branching parameter of such trees is hard to keep under control; particularly in the condition of much extended nondeterministic branches, for example the SUT selects a random integer. However, the online technique can conform to the previously remarked reflex of the SUT, and select the next test stage entirely dynamically.

One of the most significant reasons for non-determinism is timing, specifically since the testing initiation substantially provides communication delays. For instance, if the system under test transmits out a break message after ten seconds, it can be translated into testing harness in fact receives the message only after ten second because of some slowness in the testing surroundings. Identically, the inputs to the system under tests can get postponed. This is so noticeable that the UI for CQ supplies a subject for adjusting the allowed limit for communication delay.

4.2.9 Multi Language Uphold

Although QML is the original modeling language created by CQ, the tool upholds also other languages. This multi-language upholding is developed by translating all end level models into an interior process format. This format, which has been named CQ*λ*, is practically an altered scheme, a lexically domain shape of the LISP.

Hence, all QML models are converted eventually to LISP. The way for performing this includes of the following constituents:

- A model and meta-model loading at UI
- A built-in model repository
- A parsing system for lexical languages that handle fully obscure grammars

At the very beginning, a QML model including UML state machine diagram and textual program parts are loaded into the model repository by means of the UI. Meanwhile, two meta-models are loaded in parallel: QML, and CQλ. Basically, the source codes seem in the repository as not transparent strings. The second stage is to parse them and to substitute the strings with the related syntax. Macro expansion and type checking take place at this step. Then the graph re-organizer is called with a special *command set*, which is repeated until a marked point; this leads the model to be progressively translated from a crude illustration of the QML to a mature presentation of the sound CQλ. At last, the derived CQλ model is getting linear into strings of CQλ code.

```
(define-input-port input)
(define-output-port output)
(define main
  (lambda ()
    (let* ((msg (ref (handshake input #f) 1))
           (_ (handshake (tuple output msg) #f)))
      (main))))
```

```
replace "Timer trigger" {
} where {
    Transition t;
    TimeoutTrigger trigger;
    t.trigger == trigger;
    t.trigger_cql == nil;
} with {
    t.trigger_cql := '(tuple ,CQL_Symbol("__after__") ,trigger.timeout);
};
```

4.2.10 Test Case Generation

The remaining question is how this application generates test cases. The principal algorithm is an enumerator for pretended runs of the loaded model. What makes this troublesome is that this enumerator requires being capable of simulation an *independent* model, (i.e. a model that exchange message with surroundings that has not been specialized). Originally, the enumerator supposes that a completely non-deterministic environment has been connected into the independent model. This generates limitless and extensive non-deterministic branches in the tree; since the conceptual environment can transmit. Another collection of non-deterministic branches is driven by the interior selections in the model itself, and these can be also limitless wide. Consequently, the trick is how to control a state space with unlimited many states and a limitless branching parameter.

To be able to uphold several testing methods, like transition or state coverage or boundary value analysis, this application has an interior option to contain coverage checkpoints in the middle of CQλ stage models.

A coverage checkpoint is flagged in a LISP model by an invocation to the interior checkpoint process. This indicates that the creation of coverage checkpoints can be entirely managed in the model translation level. As a matter of fact, all the several end point testing methods such as transition coverage or boundary value studies have been developed in the model transformer by means of this original checkpoint capability.

Suppose the collection of all possible paths the name T, the collection of all coverage checkpoints the name C, and symbolize the set of Booleans by B. The state space enumerator can be viewed as an oracle that develops the further functions:

$$\textbf{Sound}: \quad \mathbf{T} \rightarrow \mathbf{B}$$

$$\textbf{Coverage}: \mathbf{T} \rightarrow \mathbf{2C}$$

$$\textbf{Plan}: \quad \mathbf{T} \times \mathbf{2C} \rightarrow \mathbf{T}$$

The function Sound indicates if a loaded trace is an issue that the model could create or not, so it puts a model examiner. The function coverage computes the collection of checkpoints that should have been crossed on each running-cycle of the model that creates the given trace. Lastly, plan computes an expansion for a sound trace that can be generated by the model; trying to uncover such an expansion that it would lead to a checkpoint that is not contained in the loaded collection to be crossed. Applying these oracles, a less complex version of the online mode of the application can be expressed by the next algorithm. Initialize a variable C — which will contain checkpoints — with the empty set. Set up another variable *t* — which will include a trace — with no trace. Then iterate infinitely*:* If Sound (*t*) = false, send 'FAIL' and abort testing. Else, upgrade

C to C ∪ coverage (*t*).

Then compute tprime = plan (t,C). Whether the next action in *tprime* is an input to the SUT, stop until the time of the action and then send it. Else, only stop for a while. In any condition, if a message is received from the system under test while stopping, upgrade *t* respectively; otherwise, upgrade *t* with the message already sent Offline script generation seems simpler. Since the model should be in deterministic mode inputs, everything given back by designed plan as a test case. The common idea is to encounter a collection T of traces such that $|T|$ is small and $\cup_{t \in T}$ coverage(t) is big.

Practically, the oracles sound, coverage and others are internally around a *symbolic executor*. Symbolic execution is famous and has been exploited for test derivation and program checking. Generally

developing it demands some form of *constraint solving*, and so this industrial case study also upholds a constraint solver under the application. The data fields for the solver is the least D :

$$\mathbf{Q} \cup \mathbf{B} \cup \mathbf{S} \cup D0 \cup D1 \cup \cdots = D$$

that S stands for the unlimited collection of *symbol ,* any constraint variable in a constraint solution within this solver can beforehand suppose a numeric, Boolean or symbol value, or a tuple of an free size including such values and other tuples repeatedly. Specifically, it offers strings as tuples of integers, and records as tuples including the values of the parameter. This is a weak type of structure of the constraint solver that shows the dynamic typing in the CQλ language. The constraint solver for the tool has been developed in C++ due to it is hard to integrat in the symbolic executor itself.

4.2.11 Scalability

At least in the mode developed in CQ, MBT is computationally a severe task. Practically, both memory space and time are noticeable origin factors for designing. Some work has been done to gradually enhance the time and memory attributes of CQ.

The performance of the application does not become satisfactory when it executes in lack of physical memory and starts swap trashing. To avoid this, it exchanges proactively most of the runtime threads on physical disk. The architectural method for this is established upon another shape of *reference counting*.

5. Conformiq Qtronic Analysis

In this chapter we try to analyze the tool according to some predefined functional and non-functional requirements. Our results have been based on: more than five hundred hours of loading the tool, and engaging around 10 computer scientists with the CQ.

There are some observations regarding how to implement these experiments: The experiments entirely did out of the company site. We used both Windows and Linux machines with the strong processor and at least 500 MB RAM memory. We started to train the evaluator by lecturing them first a semi-course of "What is model based testing?" and "A brief tutorial of Conformiq Qtronic".

Then we tried to get them familiar with simple models and then ask them to explore into harder models and let us know if they have any questions or ambiguities. At this point we assumed that they are familiar with CQ and assign them to create easy to complex SUT via Qtronic Modeler and program with CAQ and test it.

5.1 Reliability of the CQ:

To compute the reliability of software, we measure the inverse, which is failure intensity. Simply we can log the hours that we run the tool and note the time of failures. Also we fed the tool with several packages which differ in the sense of size and run over the nights. Below there is a table that describe the results. Regarding to measure reliability the scenario was: We assign ever evaluator a station and ask them to measure the tool in several aspects. We did this scenario in 2 hours sections. While they are assessing the tool we sat behind them and observe them

and take note. As just that they found any bug or tool misbehavior they write it down and let us know.

Observation: The experimental results show that the CQ works fine for small scale systems. However, the number of crashes increases when the number of states increases. A solution for this issue is that breaking down a large system (in the sense of states) into smaller systems and binding these divided systems together. Another solution could be use design patterns or putting more stress on abstraction and polymorphism.

Table 1: The result of reliability inspection

#State	# hours under work	# Crashes	Times
7	500	2	113,456
18	500	7	25,57,92,183,443,447,453
29	500	9	29,224,229,370,388,433,434,438,461

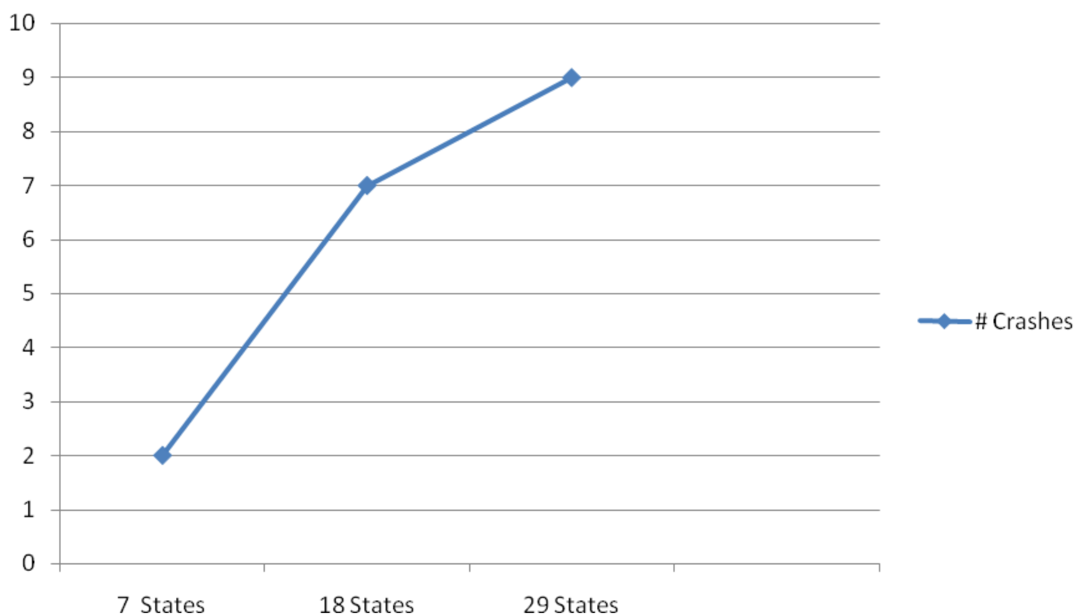


Figure 1: Relation between # crashes and # states

5.2 Credibility of test case generation algorithms

There are three main algorithms used in the CQ to create test cases:

A. Random-Walk Algorithm (R-WA): As CQ claims that:" In the random walk mode the order in which a set of test runs will be inferred is *random* and it may depend on the structure of the model."

In a simple word, R-WA is an algorithm includes passing continuous fixed steps in the random direction. Moreover, in R-WA the location of a walker at an exact time is a function that only relies on its location at the last step and a group of variable that randomly get values. By formal definition:

$$L\ (0) = L0$$

$$L\ (t + \alpha) = L\ (t) + \Omega\ (\alpha)$$

that Ω (α) explains the stochastic laws for passing steps and α is the time period from step (n -1) to (n). There are several versions of R-WA such as 1-Dimensional, 2-Diemnsional and Higher-Dimensional. Besides, there are also other types of R-WA such as: Discrete R-WA, Continues R-WA, Biased R-WA, and Unbiased R-WA. Having R-WA as the simplest way to create test case in CQ mainly have two benefits:

1. It is easy to implement.
2. R-WA Gambler rule: In any 1-Diemnsional R-WA, each state in SUT will approximately be passed unlimited times. For 2-Dimension R-WA, the rule change to the phrase that every line would be passes unlimited times.

B. Balanced Random i.e. *Markov-chain Algorithm*: As CQ claims:" This algorithm is mainly like the R-WA, however it was made to balance the random selections so that several regions of the SUT receive the same portion of load. Technically, CQ provides a random model of how the SUT is going to simulate for execution." However, in a simple word, concerning to the W-RA while the fixed distance between two points and the orientation of every inserted step relies only on the location of *L*(*t*) and independent of all previous locations. By definition:

A Markov algorithm is a set of continuous stochastic variables α1, α2, α3, α4, α5... by keeping the Markov Essential as: having the present position, the next and previous positions are independent:

$$\Pr(X_{n+1} = x | X_n = x_n, \ldots, X_1 = x_1) = \Pr(X_{n+1} = x | X_n = x_n).$$

C. Coverage Directed Algorithm: As CQ claims: "CQ applies the coverage parameters in the SUT and as chosen by tester to direct MBT straight forward. This algorithm uses higher CPU portion than the previous algorithms, however it can reaches better portion of coverage parameters by inserting less inputs."

Observation: Choosing the Markov Chain Algorithm to generate test cases in my point of view is a wise choice. In contrast, Random Walk algorithm is not such a strong or powerful algorithm. As a matter of fact it is easy to implement but it in the sense of look ahead depth, state, method and transition coverage Random Walk could not be reliable as is should be.

5.3 Efficiency

There are many ways to measure the efficiency such as: measurement of CPU resources. However, ideally, efficiency can be measured by comparing the proficiency between manual testing and testing with a tool. For this purpose, we run manual testing and MBT for the same models, and measured the number of generated test cases.
We tested three SUTs:

- ATM Machine
- Calculator
- Inventory System

There are some observations that should be taken into account before comparison:

- We installed the CQ on ten different machines from Windows and Linux with several processors and the find out the average time for installation the tool is: 0.16 hour.
- We asked four people (which they are not Conformiq employees) to start to work with CQ. The have been asked to go though the tool, get familiar to it in the level to create a simple model and create some test cases. All of the participants at least have the master degree in computer science. The average time to be able to use CQ is: 44 hours

The below chart shows the comparison between efficiency of MBT and manual testing.

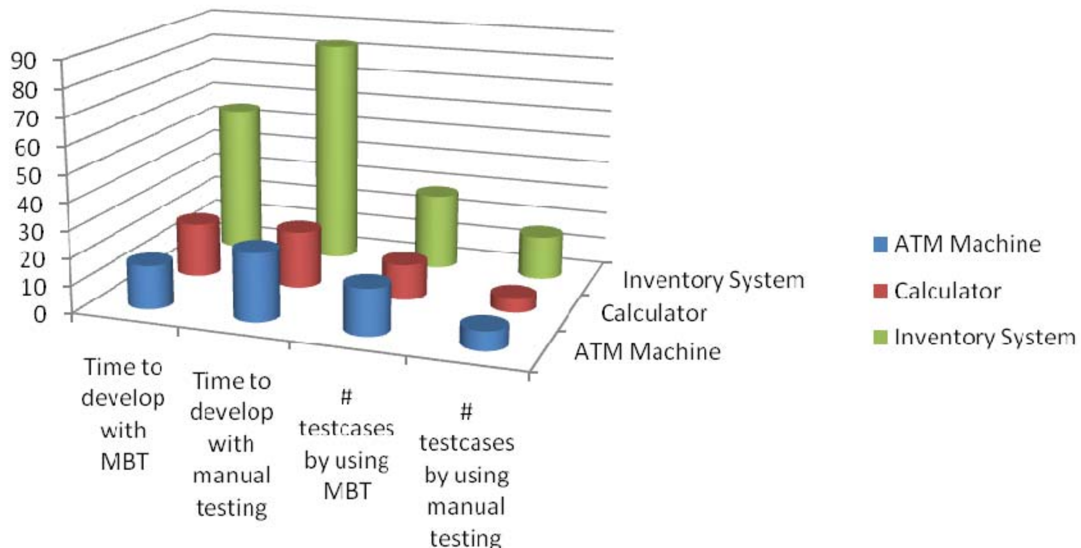


Figure 2: Comparison efficiency between MBT and manual testing

Observation: We compared there different scale SUT from small system (Calculator) to semi-Large SUT (Inventory System). The results confirm what we expected from the CQ. Small system requires less time to create rather than the larger one. However, the larger system we have more test case we have, and that is a very good point of CQ. While we have a more complex system we need more test cases to check the functionalities of the system. The other important issue that uncovered during the efficiency evaluation is that the number and precision of the test case generated automatically is higher than the ones created manually.

5.4 Maintainability

We asked the Conformiq let us use some data of the history of the tool, and they kindly prepared us a restricted data for this part. However, much of the releases are business considerations, but the supply rate of features and refactoring gives an indication of the maintainability. In the Conformiq, they have a very dynamic versioning system called *Ticket*. Every problem or observation should be reported and a unique ticket number is assigned to this report. A ticket system consists of rows of reports. Every row includes:

- Ticket Number
- Summary of report or bug
- The related component e.g. place of report
- #Version
- Respective milestone
- Type of report: Defect, Task, Enhancement …
- Severity
  - A: Blocker
  - B: Critical
  - C: Major
  - D: Normal
  - E: Minor
  - F: Cosmetic
  - R: Feature Request
  - X: Enhancement
- Priority: A range start with 1 as the highest priority and ends with 5 as the least priority
- Owner: The one who posts the report
- Date of report

The company granted us an access to a list that consists 84 rows. Our analysis is based only on these data of maintainability. Below we have gathered the results:

- Seven employees have participated in the Ticket with the portion of:

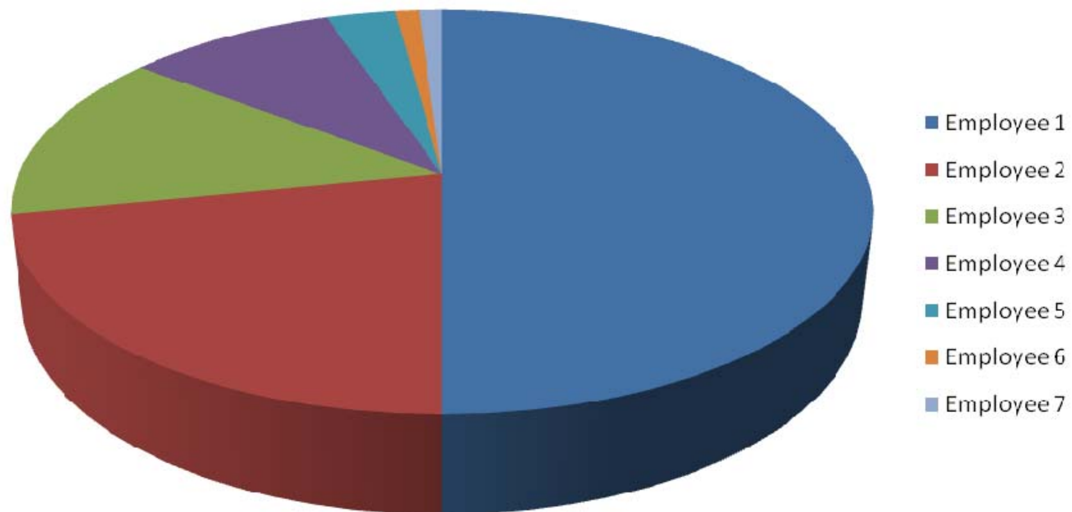


Figure 3: The portion of participation in Ticket

- The reports' location are:
  - Qtronic
  - Qtronic Adapter
  - Qtronic Packaging
  - Qtronic Compiler
  - Qtronic User Interface
  - Qtronic Modeler
  - Qtronic Manual
  - Qtronic Combiner
- There have been reported: 32 defects, 32 tasks, and 28 enhancements
- There is only one report with the severity A, however there are 8 B-severity, 11 C-severity, 20 D-severity, 3 E-severity, 6 F-severity, 18 R-severity, and 5 X-severity
- The distribution of Priority is : 39 1-priority, 29 2-priority, 6 3-priority, 3 4-priority, and 5 5-priority

Observation: The Ticket system is an inter-organizational system and it seems that the Conformiq are happy with it. In this project we are not evaluating Ticket system.

5.5 Quality of Results

Mainly, there are two ways of working on this criterion. Input to both is some kind of fault classification; possibly we can find one that is new and fresh. We already discussed how the methods work. Based on our analysis only two types of fault cannot (or hardly possible) to be detected:

- Faults happen in loops
- Faults in non-deterministic areas

We can take some buggy SUT, generally seeded with faults and create a model and start generating test cases and count which types of faults that are found mostly. We fed out three SUT with variety of bugs and amazingly CQ detected all of them except the bugs we intent fully put in the non-deterministic areas or in the loops. However, if we assigned the look ahead depth in the extreme level and asking for the creating many test cases or only finalized test cases all of them can be covered, but it takes unusual times to finish it. This, offer the tester some solutions:

- From the first moment of modeling, keep avoiding of falling into non-determinism and using loops by helping modeling technical methods.
- For archiving 100% coverage, tester should be patient and let the CQ works over nights. The results can include many useless or redundant test cases, but it can uncover all of the faults.
- In the case of time shortage, tester should get over of finding bugs in the mentioned conditions.

Observation: We have a problem here. The first situation is that the CQ never halts and it can create test cases. In this case the precision and accuracy of the results depends on the user preferences. However, if the user asks the extreme accuracy in a large system CQ will halt again. The main problem happens in the mentioned points with loops and non-deterministic areas that we even could not create test case and this is a really important drawback of CQ.

5.6 Portability

The term "Model" in CQ refers to the collection of graphical notation in the XMI format and a code that plays the role of action language in CQA format. As we described earlier, CQA is in the semi-Java notation and it has been produced by Conformiq Software. This indicates that every importable SUT should consist of CQA code. There is a trick here and that is to put the whole code in the XMI file which means that we can replaced all of declarations or function calls and etc. in the graphical notation. By using this capability, then we can import every valid XMI file into CQ, compile and use it. There are some tools that we can create XMI file by applying them such as IBM

test generator. As a smoke test we created an ATM SUT with IBM tool and imported into CQ. Then we compiled it successfully and create TTCN3 test cases based on it.

Observation: It seems that Qtronic only accept XMI file or the state machines which create in IBM CASE tools. This would restrict the user of CQ and make them only dependent on Conformiq or IBM.

6. Future of MBT & Qtronic

In this chapter we will discuss about the future of MBT and the open areas of research in MBT. In the rest, we will take look at remained component in Qtronic which are going to add in next versions.

6.1 Future of MBT

Based on the previous chapters we can find the presence of software anywhere in our life and we are rapidly finding ourselves more dependent on them. This refers to any type of system: from planes to health insurance systems. Therefore, as a matter of fact the QoS is a subject of growing significance and becoming larger interest.

For reviewing the previous chapters: we found **MBT** as a framework to verify the requirements loaded on a SUT. In MBT creating a functional model of the expected functions of the system is the beginning step of the testing. MBT has lately received concerns by growing the use of modeling. The key property of MBT is that it lets automatic test generation that looks further than creating hand-written test cases by executing them entirely. MBT creates of huge amount of test cases, containing oracles, and totally automatically from the model of needed functions. In the case that the model is correct all the test cases are correct. Furthermore, these models can be applied for declaring

coverage parameter and choosing test type. Hence a measurable assurance can be reached, and we can verify that a SUT still follows its requirements.

From a business and marketing point of view, MBT is a favorable framework to increase the excellence and efficiency of testing, and to decrease the testing costs. The present status of profession is that automatic test generation generally focuses on the performing automatic execution. Thus, a variety of industrial CASE tool are ready to reach for running automatic test execution, however these equipments never get near to the test case generation issues.

From the theoretical point of view, MBT is an ordinary expansion of formal verification methods. This type of testing is about to display that the SUT has some expected behaviors by satisfying that a model of that SUT meets these expected behaviors. Therefore, any testing is just as acceptable as the correctness of this model. MBT begins with a model, and then is about to display that the actual development of the SUT acts in accordance with this graphical notation. Because of the existing constraints of testing, like the restricted amount of test cases that can be executed, software testing may never be perfect: testing may only demonstrate the existence of failures, not their absence.

What has been gathered until now in previous chapters is a comprehensive survey of current state of MBT. However there are still some open areas in both academy and industry.

By taking a look at the current academic papers, industrial and technical report we can say most of research and works around MBT are focused on: embedded systems and telecommunication systems. However, obviously Microsoft is the most remarkable exception. Their business technical reports emphasize that testing is a significant part of software engineering, thus making this part automated in the form of MBT is critical.

Furthermore, some subjects still remained open like:

- It is very prominent for model-based testing to interact with defective and unfinished actual system, in which the requirements never can be finished, and system characteristics are continually would be changed**,** section 2.2.

- The implementation of a SUT may not be a self-contained or a sudden process; instead it requires evolving steps: it is repeatedly implemented in growth. Several versions would release, and it would have many connectors to other systems. Model-based testing has to have capabilities to deal with such problems, section 2.3.

- Another significant issue is how to acquire the formal presentation for model-based testing, and how to mix with other system background information, specification, and presentation. This problem is made worse by the assumption that there is no general agreement regarding appropriate language for displaying of the model of SUT, section 2.2.

- A specific problem is the matter of scalability in MBT, section 2.4.1.

- In order to make model-based testing applicable we have to put it in software development cycle, section 2.2.

- By evolving the software only uncovering the errors would not satisfy the tester, however tester demands for diagnosing errors. Integrating model-based diagnosis (MBD) into model-based testing may be helpful in future, section 2.3.3.

- Several types of formalism and languages are applied as the platform for model-based testing, like:

  FSM,
  CSP + CASL,
  ADT
  MSC Spec#

  A majority of these platforms includes formal semantics and syntaxes. In contrast, the less language-like platforms are more applicable in industry like Unified Modeling Language. It seems that the demand for accurate semantics for UML specifically in dynamic behavior section makes the usage MBT more advantageous, section 1.1.2

- Significant, unanswered issues are how to choose appropriate tests, how to assess the excellence of the choice, time to finish testing, how to determine the quantity of the left risks, and eventually how to infer about the excellence of the tested SUT. These obstacles are getting more important in model-based testing, because the amount of provided tests can be unlimited, section 2.3.3.

- To show the differences among several kind of model based techniques and CASE tools it is helpful if variety of benchmarks would be available, chapter 3.

- It has not determined yet weather that it is good that tester knows the complexity of applied formal techniques, or it has entirely got hidden in the CASE tool, section 2.3.3.

To offer an interesting research topic in MBT, we finish this thesis with an argument about the most challenging subjects in MBT:

1. **Assessment for coverage criteria and test excellence**: MBT algorithms can create many test suites; however there is no technique still to make a comparison among the excellence of different test cases.

2. **Test goals and use case administration**: usually, it would be essential to direct or administrate MBT in the way that the particular or untouched sections of the system can be tested.

3. **Combining distinct models:** several test generation techniques perform for specific facet of functions. For actual SUT many facets should be tested simultaneously: Transitions, checkpoints, states, control and data flows, etc. This needs combination of the related modeling methods, and the test generation formalisms.

4. **Modeling test interwork description (IWD)**: For the performing of tests, the tester is linked to the system via a specific IWD. The function of this IWD should be described

when test cases are created. Study on several types of IWD and their effects on test creation and observation is interesting.

5. **MBT of non-functional requirements**: Majority of notions, techniques and CASE tool in MBT have been specialized to testing of functional requirements. MBT of other quality properties, such as performance, usability, reliability almost indicated to non-functional requirements, is a hoping area of study.

6. **Encouraging model based testing in industry**: To make the industry progress in applying model base testing, it seems essential to design methods in industrial scale size that they are coping with the correct degree of expectation, and that information from case studies will use the further version of model based testing technique and CASE tools.

6.2 Remained Part of Qtronic

Conformiq is advancing fast to reach its mission as to be a CASE for "State of Art in MBT", however it still needs some feature to add or more attention. There are some features that are going to be added in future such as:

- Improving the usability issues and make the tool more user friendly
  - Adding "Print" option to both Qtronic Modeler(QM) and CQ
  - Adding "Replace" option for replacing the code and capture in QM
  - Facilitating creation of a new project
  - easier analysis of generated test assets
  - better integration between modeling tools and Qtronic
  - Etc.

- Improving the test case generation algorithm
  - Refine the algorithms in order to check and compile the model faster
  - Refine the algorithms in order to create test case faster
  - Refine the algorithms in order to put more heuristics for satisfying more test coverage criteria

- Putting the option of importing use cases except importing the model to follow and check the specific scenario instead of checking the model entirely

- Creating a plug-in for eclipse IDE

References

[1] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 00)*, York, UK, October 2000.

[2] K. Agrawal and James A. Whittaker. Experiences in applying statistical testing to a real-time, embedded software system. *Proceedings of the Pacific Northwest Software Quality Conference*, October 1993.

[3] Larry Apfelbaum and J. Doyle. Model-based testing. *Proceedings of the 10th International Software Quality Week (QW 97)*, May 1997. Copyright © 2001 The Authors. All rights reserved. This paper appears in the Encyclopedia on Software Engineering (edited by J.J. Marciniak), Wiley, 2001 Ibrahim K. El-Far and James A. Whittaker: Model-Based Software Testing 1999.

[4] Alberto Avritzer and Brian Larson. Load testing software using deterministic state testing." *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA 1993)*, pp. 82-88, ACM, Cambridge, MA, USA, 1993.

[5] Alberto Avritzer and Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9): 705-715, September 1995.

[6] F. Basanieri and A. Bertolino. A practical approach to UML-based derivation of integration tests. *Proceedings of the 4th International Software Quality Week Europe (QWE 2000)*, Brussels, Belgium, November 2000.

[7] Boris Beizer. Black-Box Testing: *Techniques for Functional Testing of Software and Systems*. Wiley, May 1995.

[8] Robert V. Binder. *Testing object-oriented systems*. Addison-Wesley, Reading, MA, USA, 2000.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language*. Documentation Set, Version 1.3, Rational Software, Cupertino, CA, USA, 1998.

[10] Tsun S. Chow. Testing design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3): 178-187, May 1978.

[11] James M. Clarke. Automated test generation from a behavioral model. *Proceedings of the 11th International Software Quality Week (QW 98)*, May 1998.

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott. Model-based testing of a highly programmable system. *Proceedings of the 1998 International Symposium on Software Reliability Engineering (ISSRE 98)*, pp. 174-178, Computer Society Press, November 1998.

[13] Alan M. Davis. A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, 31(9): 1098-1113, September 1988.

[14] K. Doerner and W. J. Gutjahr. Representation and optimization of software usage models with non-Markovian state transitions. *Information and Software Technology*, 42(12):815-824, September 2000.

[15] A.G. Duncan and J.S. Hutchinson. Using attributed grammars to test designs and implementations. *Proceedings of the 5th International Conference on Software Engineering (ICSE 1981)*, San Diego, March 1981.

[16] Ibrahim K. El-Far. *Automated Construction of Software Behavior Models*. Master's Thesis, Florida Institute of Technology, May 1999.

[17] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6): 591-603, June 1991.

[18] Ilan Gronau, Alan Hartman, Andrei Kirshin, Kenneth Nagin, and SergeyOlvovsky. *A methodology and architecture for automated software testing*. IBM Research Laboratory in Haifa Technical Report, MATAM Advanced Technology Center, Haifa 31905, Israel.

[19] Jonathan Gross and Jay Yellen. *Graph theory and its applications*. CRC, Boca Raton, FL, USA, 1998.

[20] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3): 231-274, 1987.

[21] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, August 2000.

[22] Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae and Hasan Ural. A Test Sequence Selection Method for Statecharts.*The Journal of Software Testing, Verification & Reliability,* 10(4): 203-227, December 2000.

[23] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[24] Alan Jorgensen and James A. Whittaker. An API Testing Method. *Proceedings of the International Conference on Software Testing Analysis & Review (STAREAST 2000)*, Software Quality Engineering, Orlando, May 2000.

[25] Paul C. Jorgensen. *Software Testing: A Craftman's Approach*. CRC, August 1995.

[26] http://www.agedis.de/documents March 2007

[27] http://www-306.ibm.com/software/rational/ March 2007

[28] http://www.objecteering.com/ March 2007
[29] http://www.gentleware.com/ March 2007

[30] http://modeling.telelogic.com/ March 2007

[31] http://autofocus.in.tum.de/index-e.html March 2007

[32] http://www.mathworks.com/products/simulink/ March 2007

[33] http://chacs.nrl.navy.mil/personnel/heitmeyer.html March 2007

[34] http://www.telelogic.com/products/tau/ March 2007

[35] http://www.telelogic.com/products/tau/ttcn/index.c March 2007fm

[36] https://www.goldpractices.com/practices/mbt/ March 2007

[37] http://research.microsoft.com/fse/asml/ March 2007

[38] Grieskamp W., Gurevich Y., Schulte W., Veanes M., **Generating Finite State Machines from Abstract State Machines**, Proceedings of ISSTA 2002 International Symposium on Software Testing and Analysis (July 2002).

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

[40] Friedman G., Hartman A., Nagin K., Shiran T., **Projected State Machine Coverage for Software Testing**, Proceedings of ISSTA 2002 International Symposium on Software Testing and Analysis (July 2002).

[41] Harry Robinson. Graph theory techniques in model-based testing. *Proceedings of the 16th International Conference and Exposition on Testing Computer Software (TCS 1999)*, Los Angeles, CA, USA, 1999.

[42] http://verify.stanford.edu/dill/murphi.html March 2007

[43] http://projects.csail.mit.edu/mulsaw/ March 2007

[44] http://www.reactive-systems.com/ March 2007

[45] http://www.mathworks.com/products/stateflow/ March 2007

[46] https://www.thedacs.com/databases/url/key/2399/2529 March 2007

[47] http://www.telelogic.com/products/tau/ttcn/index.cfm March 2007

[48] http://www.teradyne.com/ March 2007

[49] http://www.inrialpes.fr/vasy/cadp.html March 2007

[50] http://fmt.cs.utwente.nl/tools/torx/introduction.html March 2007

[52] http://www.t-vec.com/Home.asp March 2007

[53] http://chacs.nrl.navy.mil/personnel/heitmeyer.html March 2007

[54] http://www.mathworks.com/products/matrixx March 2007

[55] Harry Robinson. Finite state model-based testing on a shoestring. *Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999)*, Software Quality Engineering, San Jose, CA, USA, October 1999.

[56] Cem Kaner, Hung Quoc Nguyen, and Jack Falk. *Testing computer software.*2nd ed., Wiley, April 1999.

[57] J. G. Kemeny and J. L. Snell. *Finite Markov chains*. Springer-Verlag, New York 1976.

[58] S. C. Kleene. Representation of events in nerve nets and finite automata *Automata Studies*, pp. 3-42, Princeton University Press, Princeton, NJ, USA, 1956.

[59] A. Kouchakdjian and R. Fietkiewicz. Improving a product with usage-based testing. *Information and Software Technology*, 42(12):809-814, September 2000.

[60] Chang Liu and Debra J. Richardson. Using application states in software testing (poster session). *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, p. 776, ACM, Cambridge, MA, USA, 2000.

[61] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, July 1990, pages 50-55.

[62] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice & Experience*, March 1992, pages 223-244.

[63] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5): 1045-1079, 1955.

[64] E. F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pp. 129-153, Princeton University Press, NJ, USA.

[65] Jeff Offutt and Anyur Abdurazik. Generating test cases from UML specifications. *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML 99)*, Fort Collins, CO, USA, October 1999.

[66] Amit Paradkar. SALT: an integrated environment to automate generation offunction tests for APIs. *Proceedings of the 2000 International Symposium on Software Reliability Engineering (ISSRE 2000)*, October 2000.