# *Model-Based Testing of Distributed Systems*

Technical Report 2008-548

Ahmad Saifan, Juergen Dingel

School of Computing
Queen's University
Kingston, Ontario, Canada
{saifan,dingel}@cs.queensu.ca

September 2008

**Abstract**

This paper provides an overview of Model-Based Testing (MBT) and its activities. A classification of MBT based on different criteria is also presented. Furthermore, several difficulties of MBT are highlighted in this paper. A survey that provides a detailed description of how MBT is effective in testing different quality attributes of distributed systems such as security, performance, reliability, and correctness is given. A comparison between different MBT tools based on the classification is also given at the end of this paper.

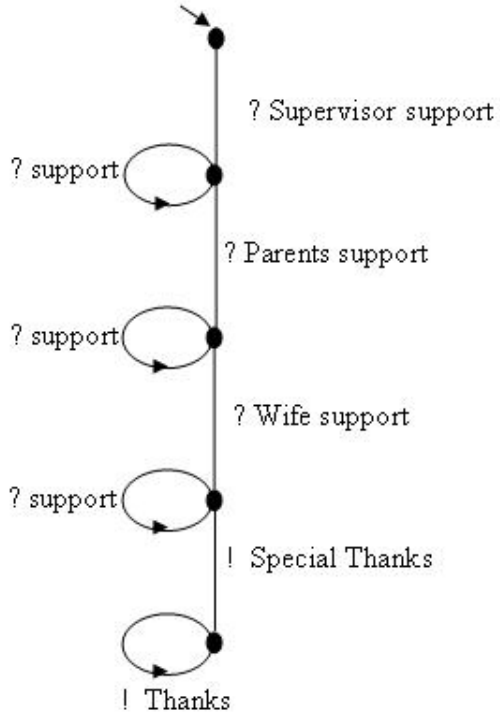
# Contents

# List of Figures



# List of Tables

# Acknowledgments




I would like to express my gratitude to all those who gave me the possibility to complete this depth paper. I am deeply indebted to my supervisor Dr. Juergen Dingel whose help, by stimulating suggestions and encouragement helped me all the time in writing this paper. I would like to give my special thanks to my parents, my wife, and my son whose patient love enabled me to complete this work. Special thanks to my brother Amjad for helping me in editing this paper. Also a special thanks to my friend Jonathan.

# 1 Introduction

Software quality is an important issue that all developers of software systems want to achieve. It currently attracts a lot of attention since software is everywhere and affects our lives on a daily basis. It is used for example in airplanes, telecommunication systems, consumer products, financial systems, administration systems, etc. So it is almost impossible to go through a day without coming in contact with software. Moreover, the size and complexity of software systems is growing dramatically, especially of distributed systems software, which typically consists of a number of independent components running concurrently on different machines that are interconnected by a communication network.

Testing plays an important role in improving the quality of software systems. It is used to check several quality attributes that are of growing concern such as correctness, performance, security, reliability of software systems.

Because of the increasing size and complexity of software, people have started to use models for many years to facilitate the development of complex software. For example, they have used models in the requirements elicitation and analysis to help understand the requirements and allow for a more detailed specification of how the system should behave. For instance, Broy et al [22], presented a method to develop models systematically from requirements. Furthermore, developers use models in the design phase, for example in his book, Gomaa [36] shows how to design concurrent, distributed, and Real-Time applications using UML. Liu et al [53] use models to design complex control systems. In addition, others use models to generate code, for example, the Altoval UModel® 2008 tool [6] is used to create Java, C♯, or VB.NET from UML models. In general, developers have largely applied models to selected elements of the development process, particularly structural and behavioral aspects in the design phase and model checking and verification in the testing phase. In general, there are at least three benefits to using models:

- *Facilitate communication*: Often, models are useful as a means of communication between the developers and the customer and between the developers themselves.
- *Enable Abstraction*: Models allow certain aspects and details of the system to be highlighted while others are ignored through abstraction.

- *Enable Analysis*: If the semantics of the model is clear, analysis can be performed; sometimes tools can be implemented to perform analysis automatically.

Additional advantages can be gained when the model is described in a formally defined language. Formal models have an unambiguously defined semantics that is more difficult to misinterpret. Also formal models allow validation of the system early. This means you can find the defects in the system as they are being developed rather than during system test, when it is in general much more costly to find and correct them. In addition, formal models can be verified mathematically, i.e it can be proved that they have or lack certain properties. Z [70], B [7], VDM [56] are examples of formal specification languages. Model-Based Testing is an attempt to leverage these models to improve software quality by detecting defects more efficiently.

The first part of this paper provides an introduction to the activities of Model-Based testing (MBT), and it also describes a classification and the limitations of MBT. The second part of this paper provides an overview of the difficulties of testing distributed systems, and also provides a detailed description of how MBT is effective in testing different quality attributes of distributed systems such as security, performance, reliability, and correctness.

### 1.1 Organization of Paper

This paper is organized as follows: Section 2 briefly describes Model-Based Testing (MBT) and its activities; in addition, this section provides a classification and some limitations of MBT. A brief description of distributed systems and their characteristics that make the process of testing them so difficult will be presented in section 3. Section 4 discusses some quality attributes of distributed systems that MBT has been used for. Section 5 presents some MBT tools. Section 6 presents conclusions and future work.

## 2 Model-Based Testing

### 2.1 The Model

El-Far and Whittaker [33] define Model-Based Testing (MBT) as an "approach in which common testing tasks such as test case generation and test result evaluation are based on a model of the application under test". During the software testing

process the testers use a model that is built from the requirements of the system in order to describe the application behaviors. Moreover, they use the model to automatically generate test cases. As described in DACS (The Data and Analysis Center for Software) [5], this model can be presented in terms of, e.g., the input sequences accepted by the system, the actions, and the outputs performed by the system. Since the model is a description of the application behavior, the model should be understandable by all testers, even if they do not have any experience or knowledge in the application domain, or they do not know what the system does. Moreover, the model should be precise, clear, and should be presented in a formal way. Section 2.4.1 will explain the process of building a model in more detail.
In general, one advantage of using models in testing as presented by Robinson [63] is that once the models have been built and assuming tools are available, the testing of the application can be performed more quickly and automatically. Another advantage is that, when the application and the test evolve, it is often sufficient to update the model incrementally with the corresponding changes.

## 2.2 Software Modeling Notations

In software testing there are several techniques to model the application behavior. In other words, there are several notations to describe the behavior of a system as a model. In [33], the authors present the most appropriate notations for testing. Some of these notations are: Finite State Machines (FSM) [18], Statecharts [40], UML [4], Markov chains [48], and Petri nets [62]. Table 1 presents these notations and in which kinds of applications they are suitable as suggested by El-Far and Whittaker [5]. Furthermore, the table describes the advantages and disadvantages of all of these notations. For more information about these models, see [33, 82]. Additional software modeling notations will be listed in Section 2.5.

## 2.3 Choosing the Software Modeling Notation

Before starting the process of MBT, the tester needs to make a decision which software modeling notation is suitable for testing the system. Or which kinds of software modeling notations are appropriate for testing the system. Today, there is no software model that fits all purposes and intents. El-Far and Whittaker [33] present several issues that the tester should be aware of when choosing a model to test an application. The first issue is the system type. For example, as we can see from Table 1 if the system to be tested has individual components and they can be modeled using state

| Software Modeling Notation | Kind of application suggested to model | Advantage | Disadvantage |
|---|---|---|---|
| FSM | State-rich systems (e.g., telephony systems) | -Easy to use<br>-Easy to generate test cases from the model because there are several graph traversal algorithms | Nontrivial to construct complex systems which have large state spaces |
| Statecharts | -Parallel systems with individual components capable of being modeled by state machines<br>-Few states, systems with transitions caused by external conditions, or user inputs | -Easier to read than FSM<br>-Allow state machine to be specifed in hierarchical way<br>-Its structure involves external conditions that effect whether a transition takes place from a particular state which reduces the size of model in many situations | Nontrivial to work with, and modeler need time training upfront |
| Sequence Diagram | Suitable for modeling a real time systems, OO systems | -Modeling complex sequences of interaction between entities<br>-Permit a lot of useful information to be shown at the same time | -Produced only with dynamic analysis, cannot be used for evaluating the adequacy of testing<br>-Cannot cover all possible aspects of interactions |
| Markov chains | Used with systems that need statistical analysis of failure | It can support estimation measures such as reliability and mean time to failure | -Require many mathematic operations, and also require a knowledge of probability theory |
| Petri Net | Suitable for concurrent as well as asynchronous behavior | Intuitive, mature notation with lots of tool support | -Poor support for modeling large and complete systems<br>-Lack of support for hierarchy |

Table 1: Selection of software modelings notations

machines, then statecharts are a reasonable solution for modeling. The second issue is the level of expertise of the people who will work with the models and the availability of tools that support MBT activities. The third issue has to do with the use of models during other part of the development. Sometimes organizations use a model to specify and design the requirements of the system, so it is better to use this kind of model also for testing, so that all teams understand the model, which makes the communication between all teams and testers easier.

## 2.4 Activities of MBT

In MBT there are several activities to test the System Under Test (SUT) see Figure 1. We will talk about these activities in more detail in the following subsection.

The process begins by building an abstract model from the requirements (see

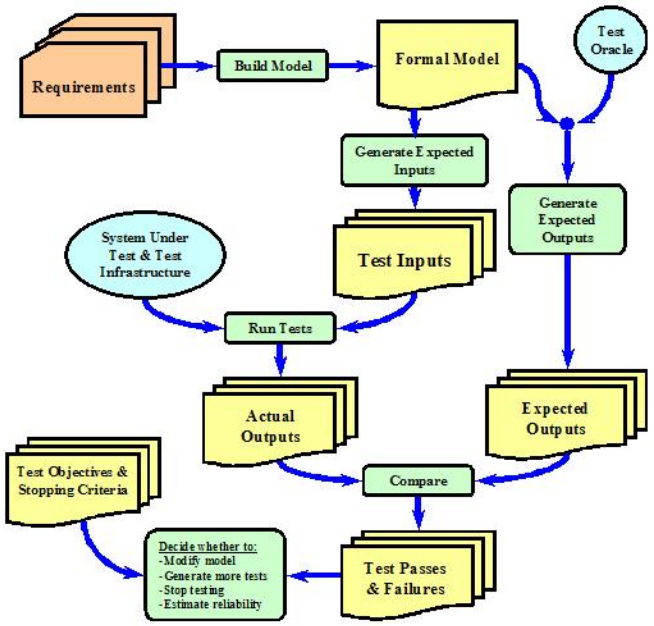



Figure 1: Model-Based Testing activities [5].

Section 2.4.1), and describing it by, for instance, one of software modeling notations presented in Table 1. The second activity is generating test cases from the model (see Section 2.4.2). The specification of test cases may include the expected outputs. Other expected outputs come from the test oracle [1]. The third activity of MBT is to run the test against the implementation of the system (see Section 2.4.3). So after generating test cases, they have to be translated into executable test scripts. These should be written in a very efficient way, since they can be used for as long as the software needs more testing. The last step, which is most important and difficult in MBT, is conformance testing (see Section 2.4.4). This activity is used to compare the actual outputs with the expected outputs. Bugs are identified in the system from the failures of the test cases. After that, the tester should decide whether to generate more test cases, modify the model, or stop testing, etc.

[1] an oracle could be, for example, a competing product or a previous version of the software

### 2.4.1 Build the Model

Building the model is the most important step in MBT, because all the next activities of MBT depend on the model. The more complete the model is, the more effective the test cases will be generated from that model. In other words, once we have an adequate model for the system, the test cases that are generated from this model will be more effective. To develop an adequate model for the application behavior, the tester needs to understand the system and its environment. Different guidelines that can be used to improve understanding of the system under test are suggested in [33, 32, 69], for example, determine the components of the system, explore target areas in the system, gather relevant useful documentation, etc. See [33, 32, 69] for more guidelines. Building the models is not easy. It needs skill, experience, and imagination from the tester in order to build a good model. Before we talk about how to build the model, we have to know what the characteristics of a good model are. El-Far [32] presents the properties that the model in general should have to be a good model. These are:

- The models should be as abstract as possible without missing important information. Prenninger et al in [60] presented different abstraction techniques that are applied in the literature on MBT. These techniques are functional, data, communication, and temporal abstractions. For more information see [60].
- The models should not contain information that is redundant or not related to the purpose of testing.
- The models should be readable to all testers.
- It should be easy to add, delete, and modify model information.

As an example of a simple model, Figure 2 presents an FSM model for a simple phone system. The nodes are the states of the system such as, On-Hook, Busy, Dial Tone, etc. Arcs are the actions that the user can perform or the inputs of the system such as, Hang Up, Pick Up, Dial/Party Ready, etc. The initial state of this system is On-Hook, and the final state is also On-Hook.

Whittaker [79], presents the general procedure to build the model, see Table 2. This procedure is suitable for many kinds of software models, such as a FSM or a Markov chain.

There are many notations used to express models in the literature see Table 1. These models are used for different purposes. For example, UML Statecharts have

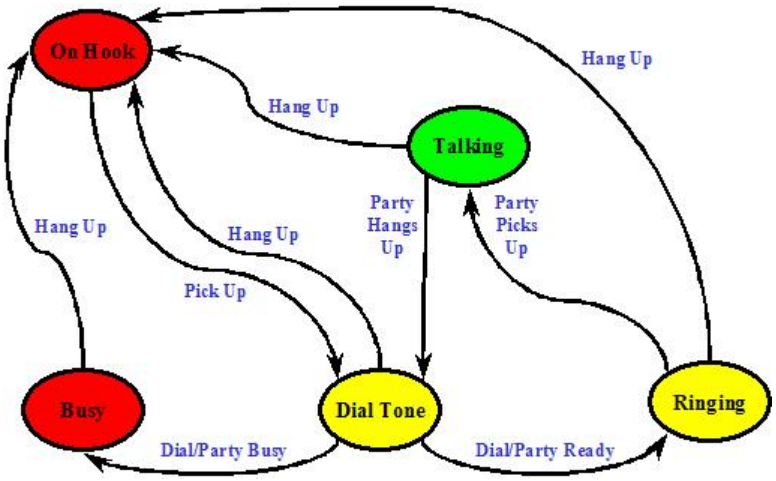



Figure 2: FSM model for simple phone system [5].

been used to generate test cases, e.g. [52, 41, 68]. Bochmann and Petrenko [16] use FSMs to specify the SUT, its implementation, and the conformance relation between the specification and the implementation. Tretmans [74] presents two algorithms for test case generation from labeled transition system (LTS) specifications. Graubmann and Rudolph[37] apply Message Sequence Charts (MSC) for the specification of test cases. Briand and Labiche[20] use Sequence Diagrams(SDs) to derive test cases, test oracle, and test driver. However, Bertolino et al [14] say that the existing approaches for building models still have some difficulties and problems. These problems are related to reducing the effort of testing, and the accuracy of the approaches. For example, state machine-based approaches[2] are generally accurate, but need expertise and effort. Also, scenario-based approaches[3] are easier to apply, but provide only partial results, since SDs cannot cover all possible aspects of interaction. However, there are other approaches that try to integrate two approaches in order to generate a complete and consistent model. For example, Bertolino et al [14] use both sequence diagrams and state diagrams in order to take the advantages from both kinds of diagrams. By combining these diagrams, a more complete and consistent model for

[2] like UML statecharts, FSM, LTS, etc.
[3] like MSC, and SD

1. List all inputs.
2. For each input, list the situations in which the input can be applied and the situations in which the input cannot be applied.
3. For each input, list the situations in which the input causes different behaviors or outputs, depending on the context in which the input is applied.

Table 2: The general procedure to build a model as presented by Whittaker [79]

testing is built, and more accurate test cases are specified without extra efforts (test cases will be discussed in the next section). The result of this approach is a sequence diagram that contains information from both the original sequence diagrams and state diagrams. Figure 3 represents the steps for this approach. At the beginning, the authors assume that sequence diagrams (SEQ) and state diagrams (ST) are available.

- In **step 1**, ST diagrams are translated into sequence diagrams (SEQ') through a synthesis process, then SEQ' and SEQ are combined together to generate a more complete model (SEQ"), since it contains information from both sequence diagrams and state diagrams.
- **In step 2** an automated tool LTSA is applied to synthesize ST' from SEQ" (many tools have been proposed to synthesis state machines from scenarios such as UBET [2], and LTSA - MSC Analyzer [76]). This step is used to ensure that SEQ" does not contain implied scenarios[4].
- If ST' contains implied scenarios, then this means that ST and SEQ are inaccurate, so in **step 3** SEQ" is modified to refine and disambiguate SEQ". Keep doing step 3 until ST' does not have any implied scenarios.
- At this point the SEQ" model represents the reasonably complete and coherent model. **In step 4** the SEQ" is called SEQrc which contains information from both ST and SEQ. After this step, the Use Interaction Test (UIT) [12] (a method used to automatically drives test cases for the integration testing phase, it uses UML sequence diagrams as a reference model) method is applied to generate test cases from SEQrc.

[4] scenarios present sets of behaviors that do not appear in the system behaviors

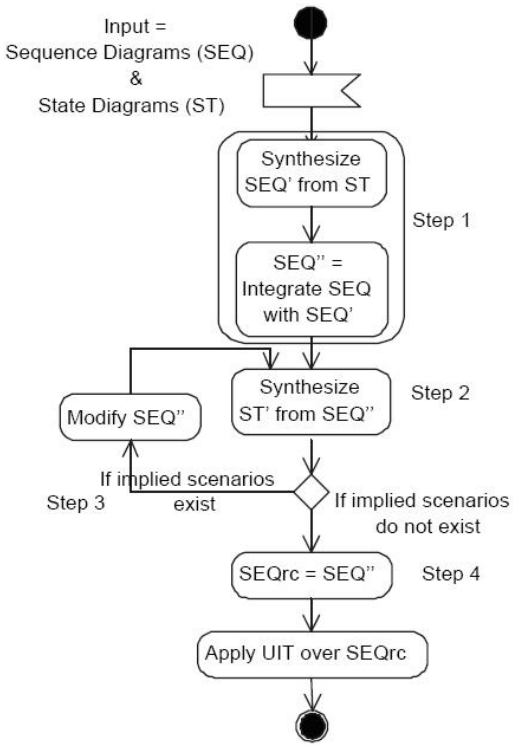



Figure 3: Process of building a reasonably complete and coherent model according to [14]

### 2.4.2 Generate Test Cases

The model that has been generated in the previous activity is used to create the test cases. Before creation is possible, the tester typically also has to specify or provide information about criteria or the purpose of the test cases, the inputs and sometimes the expected outputs of the system. The difficulties of generating these test cases depend on the model. As mentioned above, the more complete the model, the more effective test cases will be generated. Moreover, more precisely defined models make it more likely that automatic generation of test cases is possible. Without the automation, it is difficult and sometimes impossible to generate test cases for complex systems such as distributed systems. When the SUT is complex, it often means that the number of test cases is very large or even infinite. However, to improve the quality of the system, we need to select good test cases that will help the tester find as many as possible the failures in the system at an acceptable cost (cheap to derive and cheap to execute). There are several model coverage criteria in the literature that help the tester control the test generation process. Utting et al [77] discuss the most commonly used criteria. These criteria are:

1. Structural model coverage criteria: these criteria depend on the structure of the model, for example, if pre-post conditions are used to describe the model, then cause effect coverage is a common criterion. For transition-based models, coverage criteria could be for instance "all nodes", "all transitions", "all transitions pair" [5], or "all cycles".

2. Data coverage criteria: data values are partitioned into different sets. At least one representative from each set is used for a test case.

3. Requirements-based coverage criteria: requirements can be coverage criteria, when the elements of a model are associated with the informal requirements, i.e., when there is traceability.

4. Test case specification: if the test case specifications were written formally, then they could be coverage criteria.

5. Random and stochastic criteria: these criteria are suitable for environment models. The environment model represents the behavior of the environment of the SUT. Test cases are generated using probabilistic models. In other words, test cases are generated based on the probabilities that are assigned to the transitions.

6. Fault-based criteria: the most common fault-based criterion is mutation coverage [8]. In this coverage, the model is mutated. Then tests are generated to distinguish between the original and mutated model.

There are two types of test case generation: Offline and online test generation. Offline test generation means that test cases are generated and stored for example in a file before running them, so that they can be executed many times. In online test generation, test cases are generated while the system is running, because test case generation depends on the actual outputs of the SUT.

Figure 4, presents an example of a test case generated from the model presented in Figure 2. This test case presents the sequences of inputs, the states of the system after performing the actions, and the outputs of the system. The test coverage criterion used to generate this test case is "all nodes".

[5] A test set T satisfies the transition pair coverage criterion if and only if for each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$ in a statechart, there exists t in T such that t causes the pair of transitions to be traversed in sequence.

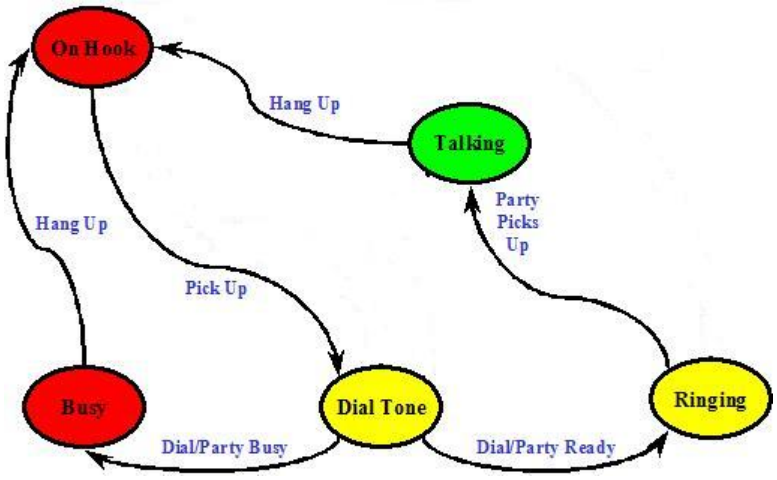



Figure 4: A test case generated from the FSM model of simple phone system adapted from Fig.2

### 2.4.3 Execute the Test Cases

In order to execute the test cases that we have previously generated from the model, we need to translate them into an executable form. Since these executable test cases can be used for as long as the software needs more testing, they should be written in a very efficient way. Figure 5 presents an executable test script for the test case presented in Figure 4. The translation will be more efficient if it is done in an automatic way. After that we apply these executable test cases to the SUT to produce the actual outputs of the system.

```
Procedure Test_Script() {
 // Here the code used to derive the software into start state

        PickUp();
        DialPartyBusy();
        Hangup();
        PickUp();
        DialPartyReady();
        PartyPicksUp();
        HangUp();
}
```

Figure 5: Executable test script

### 2.4.4 Checking Conformance

After executing the test cases and getting the actual outputs of the system, we have to evaluate and analyze the results. In checking conformance, a comparison between the actual outputs of the SUT with the expected outputs provided by the test cases is done. Ideally, this comparison is performed automatically. The outcomes of this activity are: pass, fail, and inconclusive. A test is passed when the actual outputs conform to the expected outputs. It fails if they do not conform. If we cannot make a decision at this time, the test is inconclusive. After this activity, the tester should decide whether to generate more test cases, modify the model, or stop testing and release a software system. To support this decision, the tester can, for instance, compute the reliability of the system. Reliability will be discussed in Section 4.4.
However, checking conformance is the most difficult activity in MBT. This is because it is not easy to find an automatic way for determining expected outputs (also called test oracles), especially for complex systems. The difficulty of finding a test oracle depends on the size and complexity of the system. However, the need and importance of test oracle also increases as the system becomes more complex. This problem is known as "oracle problem". Several conformance relations will be discussed in Section 4.3.2.

## 2.5 Classification of MBT

In 2006, Utting et al [77] presented a taxonomy of MBT approaches. Four of the classification criteria used are related to the model (Model subject, Model redundancy level, Model characteristics, and Model Paradigm), two to the test generation process (Test selection criteria, and Test generation technology), and one to test execution (Online or Offline). A comparison between several MBT tools based on the classification will be presented in Section 5.3. The classification criteria are:

- **Model subject**: Model could be used to present the behavior of the SUT, or its environment. A model of the SUT acts as oracle of the SUT, and is used to automatically generate test cases. The model of the environments is used to restrict the possible number of inputs to the model (acts as test selection criterion). Typically, both of these models should be abstracted. Different abstraction techniques are presented in [60, 77].

- **Model redundancy level**: Different scenarios of applying MBT may differ in the level of redundancy between modeling for test and/or for implementation. In the first scenario, the model can be used to generate test cases and code.

In this scenario there is no redundancy. However, it is not suitable for test generation, because to generate code, the model should be more detailed, but for test generation the model should be as abstract as possible. In the second scenario, the model is generated from a formal specification, and the system is implemented manually from informal requirements, so there is a redundancy.

- **Model characteristics**: Models could be nondeterministic such as most models for reactive systems. It could also incorporate timing information such as models for real-time systems. Another characteristic of a model is whether it is continuous, discrete, or a hybrid. These characteristics affect the notation used to describe the model, test selection criteria, and how test cases are generated and executed.

- **Model Paradigm**: This dimension represents the notation that is used to describe the model. Several notations are presented in [77]:

  - *State-based (Pre/Post condition) notation*: The model is a set of states that represent snapshots of the execution of the system. Each state binds variables to values. A set of operations is used to modify the variables. Each operation is defined by a precondition and a postcondition. The B notation [7] is an example of this kind of model.
  - *Transition-based notation*: A FSM is used in which the model is a collection of nodes representing the most relevant states of the system, and a set of transitions representing the operations of the system.
  - *Trace-based notation*: The systems is modeled by describing the allowable traces of model behavior over time. The representation of time could be discrete, continuous, interval, etc. MSCs are an example of this kind of model.
  - *Functional notation*: The system is described by a set of mathematical functions. This kind of model is not widely used in MBT since it is difficult to write.
  - *Operational notation*: the system is described by a set of executable processes executed in parallel. Petri nets are an example of this kind of model.
  - *Stochastic notation*: The system is described by a model containing probabilities which capture the likelihood of, e.g, the occurrences of certain events, input values or transitions. This kind of model is suitable to model

the environment of the system. Markov Chains are an example of this kind of model.

  - *Data flow notation*: The flow of data is described. Examples are Lustre [39] and the block diagrams as used, for instance, in Matlab Simulink [**?**].

- **Test selection criteria**: MBT tools can be classified according to which test selection criteria they support. These criteria are used to control the process of generating test cases, which cover the SUT. Section 2.4.2 presents these test criteria.

- **Test generation technology**: This classification represents the technology that tools use to generate test cases from the model and the test case specification. Test cases can be generated randomly, by a dedicated graph search algorithm (using, e.g, Chinese postman algorithm), through model checking, symbolic execution, or deductive theorem proving.

- **Online or offline test generation**: This dimension represents the relative timing between test generation and test execution. Online testing means that test case generation acts on the actual outputs of the SUT (test cases are generated and executed while the system is running), where offline testing means that test cases are generated before system execution.

## 2.6 Potential Problems in MBT

MBT has successfully been used for many different applications such as graphical user interfaces (e.g.,[65]) (the approach gives a reasonable coverage at a very low cost), testing phone systems (e.g,[9]), and also highly programmable systems that implement intelligent telephony services in the U.S. telephone network [30] (the test cases revealed numerous defects that were missed by traditional approaches, and the test cases are generated at low cost). Even though MBT has many advantages, it has many difficulties and drawbacks, for example:

- Skills required: The testers in MBT should have certain skills. For example, the tester should be familiar with the notation that is used to model the SUT. The modeling notation may also require that they are familiar with mathematical logic and discrete mathematics. Furthermore, they should have experience with tools, scripts, and programming languages that are used to perform the MBT tasks.

- Time required: MBT is a process consisting of several tasks, such as selecting the notation of model, partitioning system functionality into multiple parts of a model, and building the model. Each task can be very laborious and time intensive.

- State space explosion: When we have a complex system, it often means that the number of states, grammatical rules, or variable-value combinations will be very large. In addition, there will be a risk of an explosion in the size of the test cases generated from the model. Different techniques have been suggested in the literature [79] to solve this problem, e.g. optimization techniques such as: use of multiple small models, abstraction (merge complex data into a single, simple structure), exclusion (dropping information from the model without effecting the test results). However, these techniques do not always work.

- Oracle automation: as we mentioned above it is not easy to find an automatic mechanism to generate test oracles from the model (that are used to verify that the application has behaved correctly), especially for complex systems. So when the size of the system increases, the need for automation also increases. Therefore the need for automation increases the difficulty of using test oracles.

- Choice of modeling notation: As we have seen in Table 1, the author in [5] suggested guidelines for determining which notations are suitable for which kind of application. But still we need to invent a way to fit specific software models to specific application domains.

Moreover, more suitable and comprehensive theories for MBT still need to be developed. At a Dagstuhl Seminar in 2004, Brinksma et al [21] concluded the seminar with MBT research challenges. Some of these challenges are:

- It is unclear how to specify the purpose of model-based test case generation. Moreover, it is unclear how to control and guide the generation of tests.

- Additional work is needed for merging different kinds of models (Modeling notations) to build a model that allows the testers to generate test cases which cover most aspects of SUT behavior.

- Most of the theories and tools of MBT have been used to test the functional properties of the SUT. Testing with non-functional properties such as security, reliability, usability, performance, etc is still an interesting field for research.

- Integration, of techniques such as MBT, model checking, static analysis, abstract interpretation, theorem proving, etc, may be useful to be able to choose for every task the best combination of techniques.

In 2003, Robinson [64] presents different obstacles of using MBT in an industrial environment. These obstacles are:

- In most companies, software testers have less technical knowledge than the software developers. Moreover, testers are not involved in developing the system until the system has been designed and coded.
- Since the testers are involved after the system has been designed and coded, the testers only have short time to find bugs and test cases which means that they will be under pressure and they do not have enough time to build models and generate test cases etc, which could be more cost effective.
- Most software development companies do not represent the requirements formally; however, most of the developments companies use natural languages that are inherently ambiguous.

# 3 Testing Distributed Systems

## 3.1 Distributed Systems

There are several definitions of what a distributed system is. For example, Coulouris et al [29] define a distributed system as "a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing". Tanenbaum et al [72] define it as "a collection of independent computers that appear to the users of the system as a single computer". Lamport says, "a distributed system is one on which I cannot get any work done because some machine I have never heard of has crashed". Basically, this is not a definition but it characterizes the challenges that developers of distributed systems face.

Distributed systems (DSs) consist of a number of independent components running concurrently on different machines that interact with each other through communication networks to meet a common goal. In other words, in DSs the components are autonomous, i.e, they possess full control over their parts at all times. The

components, however, have to take into account that they are being used by other components and have to react properly to requests.

There are multiple points of failure in a DS. DSs could fail because a component of the system has failed. Moreover, network communication is not always successful (transmission errors) and sometimes it is not on time (delay in transmission). In real-time systems, for example, if the deadlines of the operations are not met, serious consequences may occur. Moreover, when many network messages are transmitted over a particular network, the performance of the communication may deteriorate. All of these challenges and others in DSs could affect its quality, so we need a technique to make sure that the DS is working properly without errors or failures. Testing or specifically MBT is a suitable technique. But before we talk about how MBT is effective in testing different quality attributes of DSs, Section 3.2 will give an overview of the difficulties of testing DSs in general.

## 3.2 Difficulties of Testing Distributed Systems

Testing DSs is a difficult and challenging task. It is much more difficult to test a DS than to test a sequential program. For sequential programs, we can say that the program is correct when its inputs produce correct outputs according to its specification. However, in DSs correctness of the input-output relationship alone cannot determine the correctness of DSs behavior, because the system could enter an improper state even if each process has the correct input-output relationship. Moreover, there is more nondeterminism in DSs, e.g., the delays in the message communications or the occurrence order of the events is unknown.

There are many other characteristics of DSs that make testing of these kind of systems more difficult. Typically, DSs are heterogeneous in terms of communication networks, operating systems, hardware platforms and also the programming language used to develop individual components. This means that the system should be able to run over a wide variety of different platforms and access different kinds of interfaces. Moreover, the size and complexity of DSs is growing dramatically. DSs currently attract a lot of attention and become more important in our life. DSs have been used in different critical applications in banks, hospitals, businesses, offices, etc. Furthermore, the components of DS communicate with each other through messages. Typically, these messages should arrive within a specific time window. But they could be delayed for some reason, which means messages not arrive in the same order they have been sent

(out of order delivery).

Gloshe et al [35] presented several issues that make the task of testing distributed component-based systems more complicated. Some of these issues are:

- Scalability of the test adequacy criteria: in small programs, it is easy to find adequate test coverage criteria that cover all the program behavior, such as control flow, or data flow coverage criteria. However, in complex systems such as DSs, it is not cost effective to use these criteria to test a DS. This is because of the explosion in the number of possible paths that could be taken to cover all the behaviors of the system. Moreover, sometimes it is difficult to develop the test case generation techniques for the test criteria.

- Test data generation: in order to make the set of test cases adequate with respect to coverage criteria, we need to generate test data that cover all the system behavior. However, since the number of possible paths increases exponentially in DSs, it is difficult to generate the test data that will execute all paths.

- Redundant testing during integration of components: before we test the whole system, system components are tested separately by using some adequate test criteria. However, often the same criteria to test the entire system are used, which means retesting of the components. So if the components were already tested separately, we need to know how much of extra testing is needed during integration.

- Monitoring and control mechanism in distributed software testing: since a DS is a collection of computers connected through networks, the amount of data collected for monitoring and control will be large. For this reason we need to design distributed data collection and storage mechanisms in an efficient and reliable way.

- Reproducibility of events: due to concurrent processing, asynchronous communication, and the lack of full control over the environment in DSs, specific execution behavior is often hard to reproduce.

- Deadlocks and race conditions: it is not easy to detect race conditions and deadlocks in DS. For example, harmful effect of race conditions can be hard to describe and therefore, hard to test for.

- Testing for system scalability and performance: one way of improving the performance of the system is to implement it using multiple threads. However, the system may not work when it is re-implemented with multiple threads even if it had worked when the system was implemented by a single thread that has been tested. In addition, stress testing should be done to make sure that the system performs well under high load factors. Because the system could perform very well under small loads, but fail under high loads.

- Testing for fault tolerance: to be able to test fault tolerance effectively. all circumstances in which faults can be triggered must be known. Determining all these circumstances can be very difficult

# 4 Using MBT to Improve Different Quality Attributes in DSs

As we have seen in the first part, MBT is an efficient way to test the functional properties of the SUT. But what about testing non-functional properties? Often, the system behaves correctly, i.e., it meets its functional requirements, but it violates one of its non-functional requirements. For example, it is not secure, suffers from performance degradation, deadlock, or is not robust to, e.g., unexpected user inputs, or changes in its operating environment. In this section, we will discuss how MBT activities that we have discussed in Section 2.4, can be used for in testing different quality attributes (non-functional properties) in DSs. Some of these attributes are performance, security, correctness, and reliability.

## 4.1 Performance

During performance testing, the tester should consider all performance characteristics to make sure that the DS meets the performance requirements of its users. These characteristics are: latency, throughput, and scalability. Scott Barber [11] identifies performance testing as "an empirical, technical investigation conducted to provide stakeholders with information about the quality of the product or service under test with regard to speed, scalability and/or stability characteristics". There is also other information we need to measure when evaluating performance of a particular system, such as resource usage, and queue lengths representing the maximum number of tasks waiting to be serviced by selected resources.

Testing non-functional properties of the system is different from testing functional properties. A lot of research in the literature has focused on building performance models (the first and the most important step in MBT) [58, 13, 59, 10]. Various approaches have been proposed to derive different types of performance models from software architectures (which describe the main system components and their interactions) mostly presented using different types of UML diagrams. For example, in [58], UML class diagrams and UML sequence diagrams of the software architecture are transformed into Layered Queuing Networks (LQNs) which are an extension of the Queuing Network model (QN) presented in (e.g.[51]). Stochastic Petri Nets (SPNs) are another performance model . For example in [13], statecharts and sequence diagrams are automatically transformed into generalized SPN. Stochastic Process Algebra (SPA) is another example of a performance modeling notation. For example, in [59], collaboration and statecharts diagrams are systematically transformed to SPA. A comparison of proposed approaches for transforming UML models of software architectures into performance models can be found in [10].

As an example of using MBT to test the performance of distributed systems, in [31], the authors present an approach in which the architecture design of the distributed system is used to generate performance test cases. These test cases can be executed on the middleware that was used to build the distributed application. The approach is used for early performance testing of distributed applications and proceeds as presented in Table 3.

1. Select the performance test cases (use-cases) from the given architecture designs of the distributed application.
2. Expressing use-cases in terms of operations on the middleware.
3. Generate stubs that are needed in the generation process of use-cases for those components that are available in the early stages of the development.
4. Execute the test cases, then analyze the results.

Table 3: The approach of testing performance of DSs as presented by [31]

The design of test cases in performance testing is different from the design of

test cases in functional testing. In functional testing for example, the actual values of the inputs are very important. However, in performance testing, this is not very important. To design the performance relevant test cases for distributed applications, the authors of [31] presented several performance parameters. Table 4, shows these parameters.

| | |
|---|---|
| Workload | Number of clients.<br>Client request frequency.<br>Client request arrival rate.<br>Duration of the test |
| Physical Resources | Number and speed of CPUs.<br>Speed of disks.<br>Network bandwidth |
| Middleware Configuration | Thread pool size.<br>Database connection pool size.<br>Application component cache size.<br>JVM heap size.<br>Message queue buffer size.<br>Message queue persistence. |
| Application specific | Instructions with the middleware<br>- use of transaction management.<br>- use of the security service.<br>- component replication.<br>- component migration.<br>Interactions among components<br>- remote method calls.<br>- asynchronous message deliveries.<br>Interactions with persistent data<br>- database accesses. |

Table 4: Performance parameters [31]

To check the feasibility of performance testing in the early stages of software development and the efficacy of their approach, the authors [31] performed an experiment based on Duke's Bank application[6] [17, Chapter 18]. In this experiment, the authors tried to compare the latency of the real implementation of the application

[6] an application presented in the J2EE tutorial, consists of 6,000 lines of Java code

with the latency of the test version of the same system (which is made out of the early available components) based on specific use case, while varying the number of clients. In the first phase of the approach, a sample use-case relevant to performance from the software architecture that describes the transfer of funds between two bank accounts (see Figure 6) is selected. In order to map the use case to the middleware, in the second phase, the use case with some necessary information is manually augmented. For example, "the transfer service is invoked as a synchronous call and starts a new transaction for the interaction between the client and the application. As for the interaction between the application and the database, we specify the four invocations (update the balances of the two accounts and recording the details of the corresponding transactions) are synchronous calls that take place in the context of a single transaction and embed the available SQL code; the database connection is initialized for each call" [31]

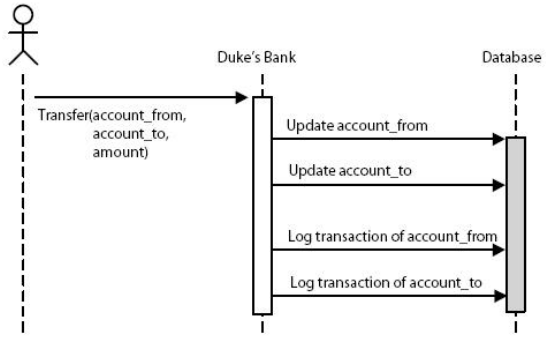



Figure 6: A sample use case for the Duke's Bank [31]

In the third phase the test version of the Duke's Bank application is developed by implementing the needed stubs in order to realize the interactions for the use cases. After that, the real implementation of the system and the test version were executed to measure the latency of the test cases. To execute these systems, a workload with increasing number of the clients starting from 1 to 100 (presenting the performance parameter of test cases) is generated. A workload generator is implemented and the persistent data is initialized in the database. The workload generator is able to activate a number of clients at the same time and takes care of measuring the average response time. For persistent data, a case was considered in which a client withdraws money from his account and deposits it in another account which is the same for all clients. Both the implementation and the test version are executed for

the increasing number of clients and measured the average response time for the test cases. Each experiment is repeated 15 times. As a result, the authors of [31] found that latency times of the application and the test version are very similar. The result of this experiment suggests that this approach is suitable for performance testing of distributed application in the early stages of software development. However, more experiments using other distributed applications need to be conducted before the general viability of this approach can be concluded. In particular, experiments using different use cases and different kinds of middleware and databases are necessary.

## 4.2 Security

The second quality issue of DSs that we are discussing is security. The use of the internet to exchange critical and personal information has increased the need for secure systems. There are different meanings of security as specified in [78]. Some people refer to security in terms of computer viruses or hackers attempting a denial-of-service attacks over the Internet. Others refer to security in terms of authentication, authorization, or encryption. As has been described in [45], taking security concerns into consideration during the early stages of software development (e.g., design and requirement phases) not only can save a lot of effort, but might even be necessary for building a secure system.

A lot of research in the literature has focused on building security models. There are various approaches extending UML diagrams to specify or to model the security requirements of the system. For example, Jürjens presents *UMLsec* as an extension of UML diagrams to specify the security requirements [46, 47]. These security requirements are inserted into the UML diagrams as stereotypes with tags and constraints. *UMLsec* is also used to check whether or not the security requirements are met by an implementation. Moreover, it can be used to find violations of security requirements in the UML diagrams. Hussein and Zulkernine present a framework for specifying intrusions in UML called *UMLintr* [44, 45]. In this framework UML diagrams are extended to specify security requirements (intrusion scenarios). One of the big advantages of using this framework is that the developers do not need to learn a separate language to specify the attack scenarios. Lodderstedt et al [54] presented a modeling language for the model-driven development of secure distributed systems as an extension of UML called *secureUML*. This language is based on Role Based Access Control (RBAC). RBAC is a model that contains five types of data: users, roles, objects, operations, and permissions. *SecureUML* can be used to automatically

generate complete access control infrastructures.

There are two categories for testing security in general as presented by Mouli [24]:

1. Security Functional Testing (testing to establish the conformance): used to test the conformance between the security function specification expressed in the security model and its implementation.

2. Security Vulnerability Testing (testing to find violations): identification of flaws in the design or implementation that can cause violations of security properties.

An example for both security functional testing and security vulnerability testing will be given in Section 4.2.1 and Section 4.2.2 respectively.

### 4.2.1 Security Functional Testing

Blackburn et al [15] developed a model-based approach to automate security functional testing called Test Automation Framework (TAF). The model is used for testing the functional requirements of centralized systems. However, the authors said that the model is extensible enough to be used for distributed systems. In private communication, one of the authors said "it would be fairly straightforward to model distributed system relationships in TTM (T-VEC Tabular Modeler) and generate vectors that could test those relationships, provided you had a testbed/test environment designed to set up, control, and record your distributed systems environment in the manner of the test vectors that would result from such an approach." The approach involves the following steps (Figure 7 shows these steps) :

1. Build the model of security function specifications using a tabular specification language called SCR (Software Cost Reduction) [42].

2. Translate SCR specifications to T-VEC Test Specifications automatically.

3. Automatically generate test vectors (i.e., test cases) from T-VEC test specifications and perform coverage analysis.

4. Develop test driver schemas and object mappings for the target test environment.

5. Automatically generate test drivers, execute tests, and generate test report.

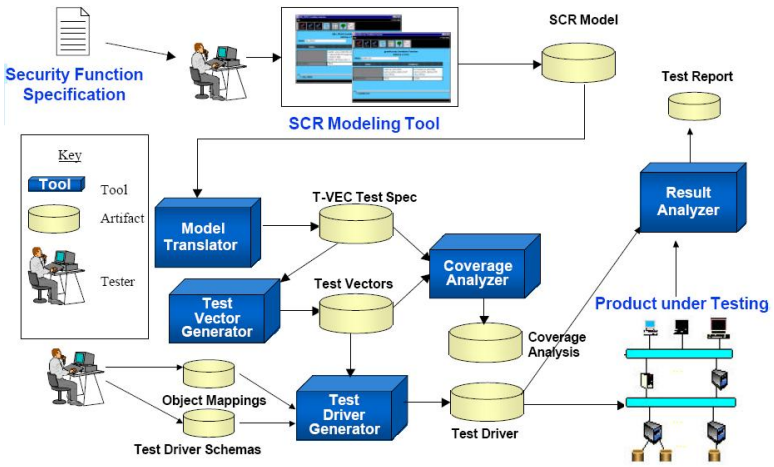



Figure 7: Test Automation Framework (TAF) [24].

In this approach, a part of Oracle8 security document [28] is used to build a functional specification model using SCR. More specifically, the security function of "Granting Object Privilege Capability (GOP)" is expressed in an SCR model. GOP as defined in Oracle8 security document [28] is "A normal user (the grantor) can grant an object privilege to another user, role or PUBLIC (the grantee) only if: a) the grantor is the owner of the object; or b) the grantor has been granted that object privilege with the GRANT OPTION."

The SCR model is a table-based model, representing the system input variables, output variables, and intermediate values as term variables. Term variables are used to decompose the relationship between inputs and outputs of the system. Once these variables are identified, the behavior of the system can be modeled.
In step 2 of this approach, the SCR tables are translated into a test case specification called T-VEC. For the GOP requirements, about 40 test specification paths were generated. In step 3, the test cases (test vectors) are generated from T-VEC test specification using particular coverage criteria called domain testing theory (which is similar to boundary testing). In order to generate test drivers that are executed against the system, the test driver generator needs the test driver schema, user-defined object mappings, and test vectors. So in step 4, test schemas are generated manually. These test schemas represent the algorithm for the test execution in the specific

test environment. In the object mapping, the authors also manually map the object variables of the model to the interface of the implementation (for Oracle 8.0.5 the interfaces are JDBC Commands, SQL Commands and Oracle Data Dictionary Views). After that in step 5, test drivers are automatically generated using the test driver generators by repeating the execution steps that are identified in the test schema for each test vector.

For the evaluation of their approach, two different test driver schemas are used (one for an Oracle test driver and another for an Interbase test driver), object mapping description, and the GOP model to test two different test environments. They found that the model executed without failure in the Oracle database driver schema, and result in test failures when using the Interbase database schema. But as described in the Interbase documentation, the failures are associated with restrictions on granting roles.

### 4.2.2 Security Vulnerability Testing

As an example of using MBT to test the system for security vulnerabilities of distributed systems, Wimmel [81] presents a new approach to find test sequences for security-critical systems. These test sequences are used to detect possible vulnerabilities that violate system requirements.

In this approach, the AUTOFOCUS tool is used to automatically generate test sequences. More specifically, mutation of the system specification and the attacker scenario is used to automatically generate the test sequences that are likely to lead to violations of the security requirements. In this approach, AUTOFOCUS is used to describe the structure and the interface of the system by using System Structure Diagrams (SSDs). An SSD presents the interaction between the system components (similar to UML component diagrams). Each component has two ports: source and destination for receiving and sending messages. These ports are connected via directed channels. Furthermore, AUTOFOCUS is used to express the behavior of the components by using State Transition Diagrams (STDs). In addition, the threat scenarios and security requirements are included to the specification of the system to generate a security-critical model. Threat scenarios represent the capability of the attacker; these scenarios are generated automatically by AUTOFOCUS based on the security attributes assigned to SSDs and STDs. There are five types of security attributes that are associated with components and channels in SSDs. These attributes

have the following meaning:

- **Critical (for components or channels)**: security-critical information is processed in the component or transmitted via the channel.
- **Public (for channels)**: the messages can be accessed and manipulated by the attacker.
- **Public (for components)**: the attacker has access to all secrets contained in the component.
- **Replace (for components)**: the component can be replaced by an attacker not knowing the secrets of the component (e.g., the attacker tries to simulate the behavior of the component without having access to it).
- **Node (for components)**: the component is an encapsulated component, to whose internals an attacker has no access.

The security attribute "critical" is also associated to the transitions and states of the STDs as appropriate. After generating the security-critical model, the authors in [81] use this model to generate test sequences in order to test the implementation of the system. These test sequences should cover all possible violations of the security requirements. In order to generate test sequences from the security-critical model, first the structure coverage criteria is needed. State or transition coverage is not suitable, because it does not take into account the security requirements. So a mutation testing approach is used. In this approach, the authors introduce an error into the specification of the behavior of the system, then the quality of test suites is given by its ability to kill mutants (distinguish the mutants from the original program).

During mutation testing, one of the critical transitions (t) of the STD of the component to be tested is chosen and then a mutation function is applied; mutation functions can be used to modify, for example, the precondition or post-condition of a transition (replace t to t$'$) in order to determine a set of mutated STDs. Next, threat scenarios are automatically generated from the mutated version of the component to be tested in order to obtain the mutated system model $\Psi'$. After that each of the system requirements $\Phi_i$ is taken, to compute a system run that satisfies $\Psi' \wedge \neg\Phi_i$ using test sequence generator. If it is successful, then mutating t into t$'$ introduced a vulnerability with respect to $\Phi_i$ and the traces show how it can be exploited. In this technique, the original specification of the components are used as an oracle (i.e, it is

used to determine the expected result).

Note that test sequences are used to test the security-critical model and not its implementation. To test the actual implementation, the test sequences should be translated to concrete test data. Concrete test data is generated from the abstract sequences by using an algorithm presented in [81]. This concrete test data is executed by the test driver that passes inputs to the component to be tested, and then check whether or not the actual outputs satisfy the expected outputs.

## 4.3 Correctness

Checking the correctness of systems is the main purpose of testing and it is the minimal requirement of most software. To check whether the system behaves correctly or not, the testers need some type of oracle. As mentioned in Section 3, correctness of input-output relationship in DS does not mean that the system behaves correctly. For example, the system may enter an improper state such as deadlock, process collision, out of order message delivery, or delays in the message communications. Section 4.3.1 gives an example of using models to check an important issue in DS which is deadlock. Section 4.3.2 will describe several criteria that are used to define the conformance relation proposed for conformance testing of DSs.

### 4.3.1 Avoid Deadlock

We could not find papers that use MBT activities as specified in Section 2.4 for deadlock detection. Instead, we will describe the use of models to ensure the deadlock free execution of a DS. For example, Chen [25] provides a control strategy to control the execution order of synchronization events and remote calls. This test control strategy is used to ensure deadlock free execution of a distributed application that consists of a set of processes (which have one or more Java thread) communicating through CORBA. All synchronization events that could lead to deadlock are sent to a central controller first which permits or defers the event based on the control strategy. Synchronization events considered in Chen paper [25] are: remote method invocation and its completion, and access to a shared object and its completion.
The following steps are used by the author [25] to construct the test control strategy:

1. Build the test constraints using static analysis. These test constraints express the partial order of synchronization events. In general, an event is represented by the name of the process and the thread this event comes from, the target

process, the name of the object on which we a call a method, the event name, the number of appearances of this event in the thread and the process, and the type of the event. Figure 8 shows the test constraints with the remote calls of arbiter system.

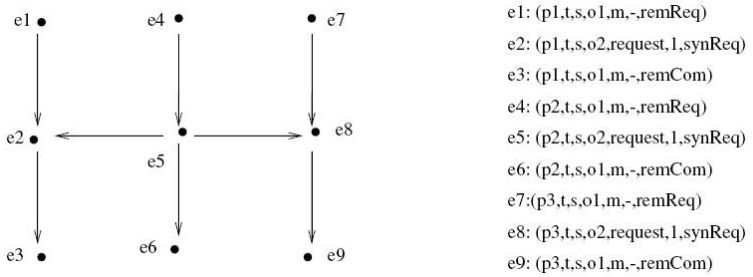



Figure 8: Test constraints and remote calls of Arbiter system [25].

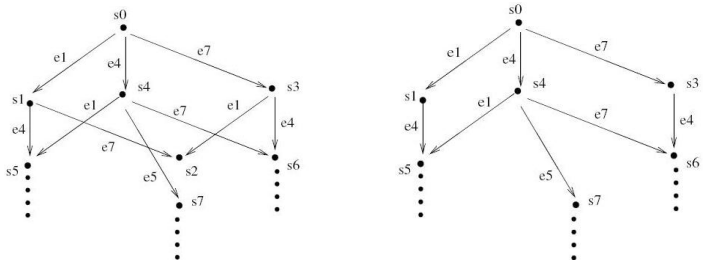
2. Choose the thread model: in the CORBA middleware, there are three thread models: thread-per-request, thread pool, and thread-per-object. If thread-per-request is used, there is no deadlock, because for every request, a separate thread is created to handle the request. So we just choose thread pool or thread-per-object as a thread model for this technique.

3. Build the test model (M) using the test constraints and the thread model from the previous 2 steps. M is a finite state automaton describing the sequence of events ( e.g., see Figure 9(a)). In M, there are two assumptions: the process p2 should obtain the server object o2 before process p1 and p3, and the thread pool size is 2.

4. Find and remove deadlock states from M to get M′ ( e.g., see Figure 9(b), s2 is a deadlock state here).

5. Controller uses the deadlock free model M′ to avoid leading the execution of the application into a deadlock state. The test controller should decide which remote calls at each moment should be executed.

An optimization technique is also presented in [25], this optimization technique is used to optimize the test model when the distributed application becomes too large. However, the approach described in [25] has the following limitations:




(a) Finite automata for Arbiter example with deadlock M

(b) Finite automata for Arbiter example without deadlock $M'$

Figure 9: Examples of arbiter system presented as finite automata [25]

- In this technique, 2 different thread models are allowed. According to the author, these thread models may not be appropriate anymore when e.g., message passing is used instead of remote method invocation using CORBA.
- The approach relies on a partial order relation of events. If this relation is incomplete, then possible deadlocks may be overlooked.
- The approach will only avoid deadlocks due to the exchange of synchronization events. Deadlocks due to other reasons will not be found.
- The approach concentrated only on the part of system behavior (test synchronization events) not the whole system behavior.

### 4.3.2 Checking Conformance

MBT is a way used to check the correctness of systems by executing test cases that have been generated automatically from the model that represent the behavior of the system. So after executing the test cases, we have already seen in the previous sections that one way to compare an implementation with a specification is to compare actual outputs (produced by the implementation) with expected outputs (as described by the specification). In this section, we will briefly present some of the conformance relations that have been defined formally in the literature. All these definitions make the simplifying assumption that both the implementation and the specification are described using some kind of labeled transition system.
The formal theory of MBT presented by Frantzen and Tretmants [34] uses labeled

transition systems[7] for modeling and an implementation relation called input-output conformance"*ioco*" to check the conformance relation between the implementation under test (IUT) and its specification. The conformance relation *ioco* expresses that "an IUT conforms to its specification if the IUT never produces an output that cannot be produced by the specification" [34]. In *ioco* theory, the implementation of the system is modeled as input-output labeled transition systems[8]. To check the conformance between the implementation and its specification, test cases are generated from the labeled transition systems using, for instance, a specific algorithm presented by Tretmants [75]. These test cases are also modeled as input-output labeled transition systems. As presented in [34], "these test cases then executed by putting them in parallel with the IUT, where inputs of the test case synchronize with the outputs of the implementations, and vice versa. An implementation passes a test case if and only if all its test runs lead to a pass state of the test case". For instance, the tool TorX [73] is using *ioco* theory to check the correctness of systems.

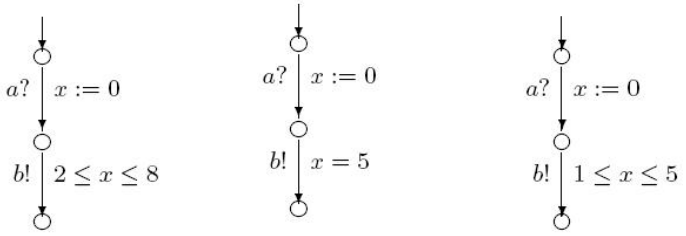
Timed input-output conformance relation (*tioco*) is an extension of *ioco* theory presented in [49]. E conforms to F w.r.t *tioco*, if for any input $\alpha$ of F, any possible output of E after $\alpha$ (including a delay) is also a possible output of F after $\alpha$. So the extension is including time delays in the set of the outputs. Figure 10(a) represents the specification of a system, where output b is produced no earlier than 2 and no later than 8 time units after receiving input a. As we can see from Figure 10(b) the implementation produces b exactly 5 time units after reception of a. So Implementation 1 in Figure 10(b) conforms to the specification in Figure 10(a). In Figure 10(c), the implementation may produce the output b after 1 time unit, which is too early. So Implementation 2 in Figure 10(c) does not conform to the specification in Figure 10(a). Algorithms are proposed to generate two types of tests: analog-clock tests which measure real-time precisely and digital-clock tests which count how many "ticks" of a periodic clock have occurred between two events.

In [57], Peled discusses additional criteria used to compare the implementation with its specification. The criteria are:

- **Trace Equivalence**: the implementation E is trace equivalent to specification F when a set of traces of E is equal to the set of traces of F.
- **Failure Equivalence**: E is failure equivalent to F when the set of failures of

[7] structure with states and transitions, states representing the system states and the transitions representing the actions that the system may perform

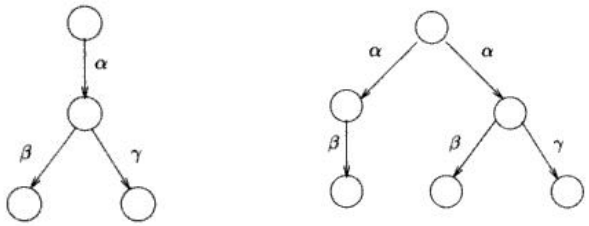
[8] it is a labeled transition system where all input actions are enabled in any reachable state




(a) Specification (b) Implementation 1 (c) Implementation 2

Figure 10: Examples of specification and implementations [49]

E is equal to the set of failures of F. Failure equivalence is a refinement of trace equivalence. Figure 11(a) and Figure 11(b) are trace equivalent but not failure equivalent since in Figure 11(a) we can always perform $\gamma$ after executing $\alpha$ but in Figure 11(b) if we choose the left $\alpha$ branch then we cannot perform $\gamma$.




(a) An automaton always perform $\gamma$ after $\alpha$

(b) An automaton where left $\alpha$ branch cannot perform $\gamma$

Figure 11: Two finite automata [57]

- **Simulation Equivalence**: Simulation equivalence ensures that every behavioral pattern in the implementation is present in the specification, and vice versa.
  Formally, E can simulate automaton F when:

  1. there exists a binary relation R between E and F such that
  2. if E′ R F′ and E′ $\xrightarrow{\alpha}$ E″, then there exists some F″ such that (F′ $\xrightarrow{\alpha}$ F″) and E″ R F″.

If E can simulate F using R relation and F simulate E using Q relation (Q does not have to be equal to $R^{-1}$), then the two automaton are simulation equivalent.

- **Bisimulation and Weak Bisimulation Equivalence**: bisimulation equivalence is a symmetric relation, which means the same simulation relation should be used ($Q=R^{-1}$).

  Formally, E is bisimulation equivalent to F iff there exist binary relation R satisfying the following:

  1. E R F
  2. for any pair of $E'$ and $F'$ and an action $\alpha$ the following hold:
     - if $E'$ R $F'$ and $E' \xrightarrow{\alpha} E''$, then there exists some $F''$ such that ($F' \xrightarrow{\alpha} F''$) and $E''$ R $F''$.
     - if $E'$ R $F'$ and $F' \xrightarrow{\alpha} F''$, then there exists some $E''$ such that ($E' \xrightarrow{\alpha} E''$) and $E''$ R $F''$.

  If E can simulate F using relation R and F can simulate E using $R^{-1}$ relation, then the two automata are bisimulation equivalent. Bisimulation equivalence is a proper refinement of simulation equivalence, and it also refinement of failure equivalence.
  Weak bisimulation equivalence is defined the same as bisimulation equivalence, but with a modified transition relation $\Longrightarrow$ that allows for the occurrences of invisible actions. E is weakly bisimulation equivalent to F when there is relation R between them such that:

  1. E R F
  2. for any pair of $E'$ and $F'$ and an action $\alpha$, where $\alpha \in$ action $\cup\{\varepsilon\}$ the following hold:
     - if $E'$ R $F'$ and $E' \overset{\alpha}{\Longrightarrow} E''$, then there exists some $F''$ such that ($F' \overset{\alpha}{\Longrightarrow} F''$) and $E''$ R $F''$.
     - if $E'$ R $F'$ and $F' \overset{\alpha}{\Longrightarrow} F''$, then there exists some $E''$ such that ($E' \overset{\alpha}{\Longrightarrow} E''$) and $E''$ R $F''$.

## 4.4 Reliability

The last quality issue of DSs that we are going to discuss is reliability. According to ANSI (American National Standards Institute) [3], Software Reliability is defined

as: "the probability of failure-free software operation for a specified period of time in a specified environment". The question here is how to use MBT to evaluate or to estimate the reliability of DSs. In the following, we will describe two approaches.

Guen et al[38], present an improved reliability estimation for statistical usage testing based on Markov chains. Statistical usage testing is used to test a software product from a user's point of view (usage model). The usage model represents how the user uses the system. In [38], Markov chains are used to represent the usage model. Markov chains usage models consist of states representing states of use and transitions labeled with usage events (stimuli). Moreover, probabilities are also assigned to all the transitions that reflect how likely a transition is to occur. The new measure of estimating reliability presented in [38] improves on the two standard reliability measures proposed by Whittaker et al. [80] and Sayre et al. [66]. It has also been implemented together in a tool named MaTeLo (Markov Test Logic). The input of the MaTeLo tool is the usage model to the software. So before using this tool, the tester should develop the usage models. MaTeLo accepts models in three different notations. These notations are: MSC (Message Sequence Chart), UML sequence diagrams, or a statecharts. However, if the tester chooses one of the first two notations, the tool converts it to a statechart. After that, the probabilities of the transitions between states are assigned using a Markov usage editor component, in order to get the design of the usage models. These probabilities can be calculated automatically based on a pre-defined distribution or can be manually specified by the user. For more information about how to calculate the probabilities of the transitions, see [80],[66], and [38]. Next, the tool checks the correctness of the usage models (e.g. the probabilities are arranged between 0 and 1, or terminal states have no outgoing transitions, etc) and then converts it into test cases. Test cases are then automatically generated based on several test generation algorithms. These algorithms are based for example on the probability of the transitions, the test cases that have minimal test steps, or on randomization. The test cases that have been generated can be represented in the formats TTCN-3 or XML. After the generation of the test cases, they are executed against the system under test and then the results are recorded. The results of the test are analyzed to estimate the reliability probability of the usage model obtained from the test runs. In [38], the authors use the following formula to calculate the reliability of the system based on the above definition:

$$\mathrm{R(t)} = (P_u)^{\mu t}$$

where $\mathrm{P}_u$ is the usage probability that no failure occurs during the software execution, and $\mu$ denotes the number of activation during a time period.

As another example of using MBT to test the reliability of systems, the author in [50], describes a tool named SpecTest. SpecTest implements and automates a model-based statistical testing methodology. This methodology uses statistical usage testing to evaluate software system reliability of complex systems using MBT. Figure 12 shows the components of the SpecTest tool.

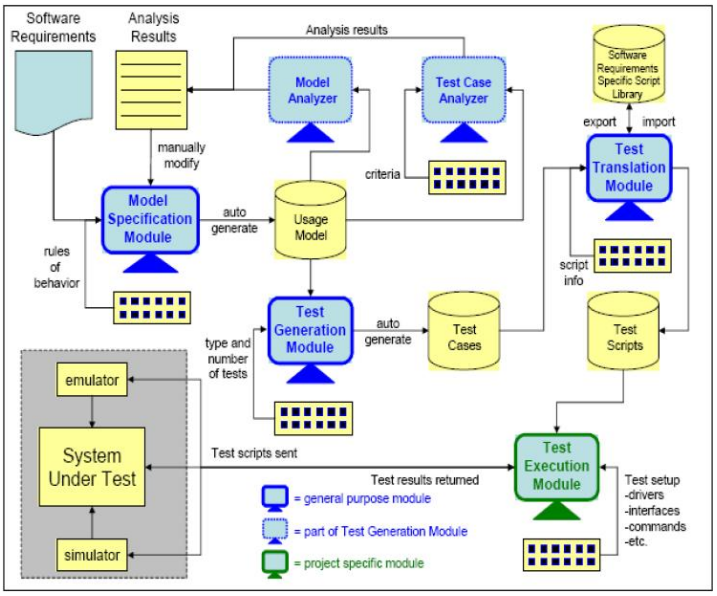



Figure 12: SpecTest tool components [50]

In SpecTest, the model specification module automatically creates a Markov chain usage model based on the system requirements. When using the SpecTest tool, the model is in the form of a table composed of states, transitions, and probabilities for each transition. To check the correctness of the usage models, a model analyzer is used. A test generation module is used to automatically generate test cases from the Markov chain model. The test generation algorithm supports state coverage and arc coverage. Test cases are described by a sequence of transitions where each transition is described using a restricted English grammar. The SpecTest tool uses the translation module to convert these test cases to executable test scripts in the test environment for the system under test. Some of these executable test scripts are automatically executed in the system under test using the test execution module. The results of these test cases (whether the test case is pass or fail) are evaluated manually. This data is used to evaluate the reliability of the system. According to the author, the

tool has been successfully applied to the US Army's Mortar Fire Control System (MFCS), a Windows XP-based application for defining and managing all aspects of an artillery mission.

# 5 MBT Tools for Testing Distributed Systems

In this section, MBT tools are discussed in detail and the summary of a detailed comparison of 7 different MBT tools is provided. TAF tool is discussed in Section 4.2.1.

## 5.1 Spec Explorer

Spec Explorer [23] is the second generation of MBT technology developed by Microsoft. The first generation was released in 2003, and Spec Explorer was generated in 2004. People in Microsoft are using Spec Explorer for testing e.g. operating system components, reactive systems, and other kinds of systems. Figure 13 illustrates the use of the Spec Explorer tool.

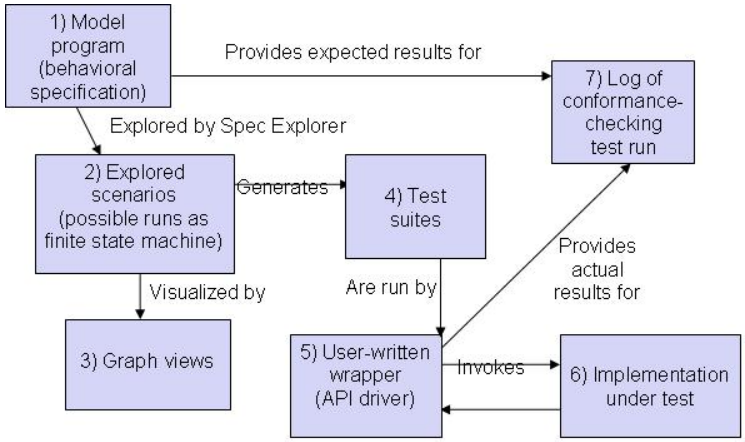



Figure 13: Steps of Spec Explorer tool [27]

In the first step of Spec Explorer, the tester builds the model program of the the system. The model program specifies the desired behavior of the system. It

can be written in any high level programming language such as Spec♯, C♯, or VB. The model program is not the implementation of the system. Typically it is more abstract than the implementation. In other words, the model program just focuses on the aspects of the system that are relevant for the testing effort. Furthermore, it represents the constraints that a correct implementation must follow. More precisely, the model program declares a set of action methods, a set of state variables, and the preconditions of these states.

In the second step of this tool, Spec Explorer explores all the possible runs of the model program and represents them as a FSM, named model automaton. A state in this automaton corresponds to a state of the model program, and transitions represent the method invocations that satisfy their preconditions. This automaton can have a very large number of transitions. To solve this problem several techniques for selectively exploring the transitions of the model program are used. These techniques as specified by the authors [23] are:

- Parameter selection: This technique limits the exploration of the model program to a finite but representative set of parameters for the action methods.
- Method restriction: This technique removes some transitions based on user provided criteria such as invocations of specific methods.
- State filtering: This technique prunes away states that fail to satisfy a given state based predicate.
- Directed search: This technique examines finite sequences of transitions with respect to user provided priorities, where the states and transitions that are not visited are pruned away.
- State grouping: This technique selects representative examples of the states from user provided equivalence classes.

The graph of the FSM can be viewed in different ways in step 3, e.g. based on how to group the similar states. In step 4, test cases are generated from the model automaton. There are two different kinds of testing in this tool: offline testing and online testing. Test cases are generated using traversal algorithms based on different test criteria. The test criteria are: random walk by specifying bound of the walk, transition coverage, or using the shortest path between the initial and an accepting state.

In step 5, test cases are executed against the system under test. To do that, an API driver can be used. The API driver (which is a program written in C♯ or

VB) does whatever is necessary in order to invoke the individual actions of the test suites in the implementation under test (step 6). In other words, the corresponding methods and classes of the API driver should be mapped to the model. This process is called object binding. We have to note here that the implementation under test uses the same names of the action methods in the model program. In step 7, the ioco conformance relation is used to compare the expected outputs of the model with the actual outputs of the implementation.

## 5.2 TorX

TorX [73] is one of the main achievements of the Côte de Resyste project [1]. Côte de Resyste is a research and development project aimed at improving the testing process using formal methods. It focusses on testing the behavior of reactive systems. The purpose of the TorX tool is to implement ioco (input output conformance) theory, as specified in Section 4.3.2.

In TorX there are four main phases that are provided automatically in an on-the-fly manner. The phases are: test case generation from the specification of the system under test, test implementation (translate the test cases into executable test scripts), test execution, and test analysis (checking the conformance between the actual outputs and the expected outputs).

In the first step of TorX, the behavior of the system under test is specified using labeled transition systems. TorX contains several explorers to explore the labeled transition system defined by specifications given in different formal notations such as LOTOS, PROMELA, or LTSA. TorX implements a test derivation algorithm that is used to generate test cases from the labeled transition systems. Test cases are generated randomly based on a walk through the state space of the specification. To maximize the chance of finding errors in the system under test, two kinds of test coverage criteria are used. Heuristic criteria provide some assumptions about the behavior of the system under test such as the length of the test case. Additionally, TorX allows the test purpose criterion, that is, of a particular part of the behavior of the system. If the test purpose is given, the test case generation algorithm guarantees that generated test cases will exercise that part of the system. In TorX, the test generation and test execution phases are integrated into one phase since test cases are generated on-the-fly during the test execution. Also, offline test case generation is available in TorX.

In TorX, there is a component called adapter, this component is used to translate the input actions of the test case that have been generated from the specification to be readable by the real implementation of the system under test. It is also used to translate the outputs that are produced by the implementation back to the output action of the test case. Then the ioco relation is used to check the conformance between the actual outputs of the real implementation and the expected outputs of the test cases.

## 5.3 AETG

The Automatic Efficient Test Generation (AETG) [26] is a tool used to generate efficient test cases from user defined test requirements. The main contribution of this tool is to reduce the number of test data needed for the input test space. It also supports n-way testing (n parameter values are covered in the test cases). However, the oracle of the test cases should be provided manually by the user. AETG has been used to generate test cases for different kinds of applications such as an ATM network performance monitoring system, and in telephone systems.

## 5.4 Conformiq Qtroniq

Conformiq Qtroniq [43] is founded in 1998, Finland. It is a product of Conformiq Software Ltd. It is used to automatically derive functional test cases from the system model. The model in Conformiq Qtronic is expressed as a collection of Statecharts diagrams with blocks of Java and C♯. In this tool test cases can be generated online or offline by using a symbolic execution algorithm. The test cases are presented as a sequence of timed messages. Then, the test cases are mapped into TTCN-3 format. There are various coverage criteria available in this tool in order to generate test cases such as state coverage, arc coverage, condition coverage, requirements coverage, etc. The expected outputs of the test cases are also automatically generated from the system model.

## 5.5 LTG

LEIRIOS Test Generator (LTG) [19] is an industrial tool developed in the laboratory of computer science of the University of Franche-Comte' (LIFC). It focuses on testing the functional behavior of smart card applications.

In LTG, the functional behavior of the application under test can be specified with Statecharts, the B notation, and UML class and state diagrams. This functional model is validated by simulating its execution using the LTG model animator. After that, test cases are automatically generated using symbolic execution of the model or by using a search algorithm. There are different coverage criteria that can be used in this tool to generate test cases. For example, if the model is presented in UML diagrams then for example all the transitions, all transition pairs criterion can be used. If B notation is used as a model of the system, then all-effect, all pair of effect can be used as a test coverage criterion. After that, test cases are automatically translated into test scripts. Each of the test cases includes its expected outputs, so the conformance is automatically computed.

## 5.6 JUMBL

J Usage Model Builder Library (JUMBL) [61] has been developed by the software Quality reliability Laboratory of the University of Tennessee. JUMBL uses statistical usage testing to evaluate the reliability of DSs. The tool does not have a usage model editor. However, it allows the user to specify the model in a variety formats e.g. The Model Language (TML), and Common Separated Value (CSV). The tool uses the Chinese post man algorithm to generate test cases from the usage model. In JUMBL test cases are presented using Test Case Markup Language (TCML) (an extension of XML). In this tool, test oracles provided by the user. The tool has been applied to scientific application software in [71, 67].

## 5.7 A Comparison Between Different MBT Tools Based on the Classification

In general, in MBT there are 4 main phases. These phases are:

1. Build the model.
2. Generate test cases.
3. Execute the test cases.
4. Check conformance.

In [77], Utting et al specify several criteria for the classification of MBT. Four of these criteria are related to phase 1, two criteria to phase 2, and one criterion to phase 3.

The classification is used to compare different MBT tools.
In [77], the authors' classification does not include a criterion for phase 4. Moreover, the classification does not include the purpose of testing. In addition, the test case notation does not used as a criterion for phase 3. Therefore, three other criteria will be added to the classification. Moreover, other tools are added to the tools that Utting et al mentioned in their paper. To summarize, the new three criteria are:

- **Checking conformance**: This criterion is used to classify how different MBT tools check the conformance between the actual and the expected outputs of the system under test. In other words, how the tools determine whether a test case is pass or fail. There are different ways to check conformance, for example, some tools use conformance relation theory. For example, TorX and Spec Explorer use ioco conformance relation. Other tools use test oracles to check conformance. Test oracle can be generated automatically by the tool or they can be provided manually by the user. Moreover, conformance can be decided by the human.

- **The purpose of testing**: As we have seen in the previous sections, systems can be tested for different purposes. For example:

  - Testing the functional properties of the system: The model in this kind of testing represents the behavior of the main functions of the system under test.

  - Testing non-functional properties of the system: As we have seen in the previous sections, we can test the performance, security, reliability, etc properties of the system. For example if we are going to test the security of the system, then the model will represent the security requirements of the system under test.

- **Test case paradigm**: This criterion is used to describe the notation of the test cases. There are different notations that can be used to describe the test cases e.g. FSM, sequence of operations, input output labeled transition systems, TTCN-3, etc.

Table 5 compares different MBT tools based on Utting et al's modified classification. Also, three other tools are added to the tools presented in Utting et al paper [77], these tools are: Spec Explorer [23], Conformiq Qtronic [43], and TAF [15].

| | | | $phase1^a$ | | | | $phase2^b$ | | $phase3^c$ | | $phase4^d$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MBT tool | Academic/ commercial MBT tool | Purpose of Testing [e] | Subject of the model | Model redundancy level | Model characteristics | Model paradigm | Test selection criteria | Test generation technology | Test case paradigm | Online/ offline | Checking conformance |
| TorX [73] | Academic | F | SUT behavior model with some environmental aspects | Dedicated testing model | Nondeterministic, untimed, discrete | Input Output Labeled Transition System | A Walk(done randomly or based on test purpose) through state space of the model | On-the-fly state space exploration | input output transition systems | Both | ioco relation |
| LTG [19] | Commercial | F | SUT behavior model | Dedicated testing model | Deterministic, untimed, discrete, and finite | UML statemachines, B notation | For Statemachines: state coverage, all transitions, all extended transitions, and all transitions pairs. For B notation: all effects, and all pair effects | Symbolic execution, search algorithm | Sequence of operation | Offline | Expected outputs assigned with each test case |
| JUMBL [61] | Academic | R | Behavior environment model | Dedicated testing model | Untimed, and discrete | Markov chain usage models | Random and statistical criteria | Statistical search algorithm | Test case markup language | Offline | Test oracle provided by user not the tool |
| AETG [26] | Commercial | F | Data environment model | Dedicated model for test input generation only | Untimed, discrete | No modeling of the behavior, just data | Data coverage criteria | N-way search algorithm | Table of inputs values | Offline | Test oracle provided by user not the tool |
| Spec Explorer [23] | Commercial | F | SUT behavior model with environment model | Dedicated testing model | Nondeterministic, Untimed, discrete | FSM | Randomly, shortest path, transition coverage | Traversal algorithms | FSM with different views | Both | ioco relation |
| Conformiq Qtronic [43] | Commercial | F | SUT behavior model with environment model | Dedicated testing model | Nondeterministic, timed | UML statemachines with blocks of Java or C♯ | State coverage, arc coverage, branch coverage, condition coverage, requirement coverage | Symbolic execution algorithm | TTCN-3 | Both | Expected outputs are automatically generated from the model |
| TAF [15] | Commercial | S | SUT behavior model with environment model | Dedicated testing model | Deterministic, untimed, hybrid | SCR table converted to T-VEC | Domain testing theory | Ttest vector generation | Test vector | Offline | The test oracle is the model itself |

Table 5: Some of commercial and academic MBT tools based on the classification

[a] Phase1: Build the model

[b] Phase2: Generate test cases

[c] Phase3: Execute test cases

[d] Phase4: Checking conformance

[e] F: Functional, S: Security, R: Reliability

# 6 Conclusion

During the MBT (Model-Based Testing) process the testers use a model that is built from the requirements of the system and its environment in order to describe the abstract behavior of the system under test. Then the model is used to automatically generate test cases. These test cases are executed in the system. After that, a comparison between the actual and expected outputs is done. We have described these major activities of MBT. Moreover, several potential problems are presented in this paper.

Distributed systems (DSs) consist of a number of independent components running concurrently on different machines that interact with each other through communication networks to meet a common goal. Testing DSs is more difficult than centralized systems. Difficulties of testing DSs are also highlighted in this paper.
Testing non-functional properties of DSs such as security, reliability, performance, etc. using MBT technique is an interesting field for research. For example, a lot of research has been done to specify the performance model of the system under test, but still we do not have a tool used to evaluate the performance of DSs. Moreover, Dias Neto et al [55], present that several types of non-functional qualify attributes of DSs such as usability, maintainability, and portability have not been tested using MBT. This paper summarizes how MBT can be used for testing different non-functional properties of DSs such as performance, security, correctness, and reliability.
As we have seen in this paper, the classification of MBT proposed in [77] is suitable to compare between different MBT tools. However, three new different criteria are added to this classification in this paper. These criteria are: test purpose, test cases paradigm, and the way the tool checks the conformance between the expected outputs and actual outputs. We have gathered information about 8 different MBT tools and compared them (4 of them listed in [77]). Some of these tools are academic and others are commercial. Moreover, some of them are used for testing the functional properties of the system, others for testing the non-functional properties of the system under test. This paper can be used to guide the reader to relevant publications in MBT. Moreover, it highlights potential problems, and how MBT can be used to test different non-functional properties of DSs. Furthermore, this research raises several interesting questions suitable for future work.

## 6.1 Future Work

The following topics could be considered for future work:

- Most of the theories and tools of MBT have been used to test the functional properties of the software under test. MBT for non-functional properties such as security, reliability, usability, performance, etc is still an interesting field for research.

- There is no technique that can be used to build different models for different quality attributes of DSs. Maybe it is difficult to do that or impossible.

- In [32], El-Far presents some of the modeling notations and discusses in which applications they are suitable as suggested based on his experience. But still we do not have specific criteria to say this kind of notation of the model is the best for this kind of system. Same remarks apply to the choice of test coverage criteria to select the test cases.

- We need additional work for merging different kinds of models to build a model that allows the testers to generate test cases which cover most aspects of software under test behavior. In [14] sequence diagrams and state diagrams are integrated to take the advantages from both kinds of diagrams.

- Detecting deadlock in DSs is difficult. We could not find a paper talking about using the MBT activities to detect deadlock in DSs. In [25], we have seen how to use models to avoid deadlock in a DS. So still we do not have a technique that uses all the activities of MBT to detect deadlock in DSs.

# References


[1] Côte de Resyste. Available at:http://fmt.cs.utwente.nl/CdR/.

[2] UBET. Available at:http://cm.bell-labs.com/cm/cs/what/ubet/index.html.

[3] IEEE standard glossary of software engineering terminology. Technical report, 1990.

[4] The Unified Modeling Language 2.0: Superstructure FTF convenience document (ptc/04-10-02). Object Management Group, 2004. Available at:www.uml.org.

[5] DACS Gold Practice Document Series. Software Acquisition Gold Practice, model-based testing. itt corporation, 2006. Available at: https://www.goldpractices.com/practices/mbt/index.php.

[6] UModel-UML tool for software modeling and application development, 2008. Available at: http://www.altova.com/products/umodel/uml_tool.html.

[7] Jean-Raymond Abrial, Matthew K. O. Lee, David Neilson, P. N. Scharbach, and Ib Holm Sørensen. The B-Method. In Søren Prehn and W. J. Toetenel, editors, *VDM Europe (2)*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 1991.

[8] Paul Ammann, Paul E. Black, and William Majurski. Using Model Checking to Generate Tests from Specifications. In *ICFEM*, pages 46–54, 1998.

[9] Alberto Avritzer and Brian Larson. Load Testing Software Using Deterministic State Testing. In *ISSTA '93, Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 82–88, Cambridge, MA, USA, 1993.

[10] S. Balsamo and M. Simeoni. On transforming UML models into performance models. In *Proc. of Workshop on Transformations in the Unified Modeling Language, ETAPS'01*, 2001.

[11] Scott Barber. What is performance testing?, 2007. Available at:http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1247594,00.html.

[12] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The cow_suite approach to planning and deriving test suites in UML projects. In Jean-Marc

Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 383–397. Springer, 2002.

[13] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analyzable Petri net models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 35–45, New York, NY, USA, 2002. ACM.

[14] Antonia Bertolino, Eda Marchetti, and Henry Muccini. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. *Electr. Notes Theor. Comput. Sci.*, 116:85–97, 2005.

[15] Mark R. Blackburn, Robert D. Busser, Aaron M. Nauman, and Ramaswamy Chandramouli, editors. *Model-based Approach to Security Test Automation*, proceding in Quality Week, June 2001.

[16] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124, New York, NY, USA, 1994. ACM.

[17] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, and Beth Stearns. *The J2EE Tutorial*. Addison Wesley, 2002.

[18] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[19] Fabrice Bouquet, Bruno Legeard, Fabien Peureux, and Eric Torreborre. Mastering Test Generation from Smart Card Software Formal Models. In *Procs. of the Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 70–85, Marseille, France, March 2004. Springer.

[20] Lionel Briand and Yvan Labiche. A UML-Based Approach to System Testing. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 194–208. Springer, 2001.

[21] Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors. *Perspectives of Model-Based Testing, 5.-10. September 2004*, volume 04371 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005.

[22] Manfred Broy and Oscar Slotosch. From Requirements to Validated Embedded Systems. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 51–65, London, UK, 2001. Springer-Verlag.

[23] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical report, Microsoft Research, Redmond, May 2005.

[24] Ramaswamy Chandramouli and Mark Blackburn. Model-based Automated Security Functional Testing. In *OMGs Seventh Annual Workshop On DISTRIBUTED OBJECTS and COMPONENTS SECURITY*, Baltimore, Maryland, USA, April 2003.

[25] Jessica Chen. On Using Static Analysis in Distributed System Testing. In *EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects*, pages 145–162, London, UK, 2001. Springer-Verlag.

[26] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatiorial Design. *Software Engineering*, 23(7):437–444, 1997.

[27] Microsoft Corporation. Spec Explorer Reference, 2005.

[28] Oracle Corporation. Oracle8 Security Target Release 8.0.5, April 2000.

[29] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

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

[31] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. *SIGSOFT Softw. Eng. Notes*, 29(1):94–103, 2004.

[32] Ibrahim K. El-Far. Enjoying the perks of model-based testing. In *Proceedings of the Software Testing, Analysis, and Review Conference (STARWEST 2001)*, October/November 2001.

[33] Ibrahim K. El-Far and James A. Whittaker. Model-Based Software Testing. Encyclopedia of Software Engineering (edited by J. J. Marciniak). Wiley, 2001.

[34] Lars Frantzen and Jan Tretmans. Model-Based Testing of Environmental Conformance of Components. In F.S. de Boer and M. Bonsangue, editors, *Formal Methods of Components and Objects – FMCO 2006*, number 4709 in Lecture Notes in Computer Science, pages 1–25. Springer, 2007.

[35] S. Ghosh and A. Mathur. Issues in testing distributed component-based systems. In *Proceedings of the First International ICSE WorkshopTesting Distributed ComponentBased Systems 1999.*, Los Angeles, CA, May 1999.

[36] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with Uml*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[37] Peter Graubmann and Ekkart Rudolph. HyperMSCs and Sequence Diagrams for Use Case Modelling and Testing. In *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 32–46, 2000.

[38] Helene Le Guen, Raymond Marie, and Thomas Thelin. Reliability Estimation for Statistical Usage Testing using Markov Chains. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 54–65, Washington, DC, USA, 2004. IEEE Computer Society.

[39] N. Halbwachs. A tutorial of Lustre. 1993.

[40] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[41] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-Based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70, New York, NY, USA, 2000. ACM.

[42] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[43] Antti Huima. Implementing conformiq qtronic. In *TestCom/FATES*, pages 1–12, Tallinn, Estonia, June 2007.

[44] Mohammed Hussein and Mohammad Zulkernine. UMLintr: A UML Profile for Specifying Intrusions. In *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006), 27-30 March 2006, Potsdam, Germany*, pages 279–288. IEEE Computer Society, 2006.

[45] Mohammed Hussein and Mohammad Zulkernine. Intrusion detection aware component-based systems: A specification-based framework. *The Journal of Systems and Software*, 80(5):700–710, 2007.

[46] Jan Jürjens. Towards Development of Secure Systems Using UMLsec. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*, pages 187–200. Springer, 2001.

[47] Jan Jürjens. *Secure Systems Development With UML*. Springer-Verlag, 2005.

[48] J. G. Kemeny and J. L. Snell. *Finite Markov chains*. Springer-Verlag, New York, USA, 2nd edition, 1976.

[49] Moez Krichen and Stavros Tripakis. Black-Box Conformance Testing for Real-Time Systems. In *SPIN*, pages 109–126, 2004.

[50] Peter B. Lakey. SPECTEST: A Model-Based Statistical Testing Implementation Tool. In *ISSRE '07: presented at 18th International Symposium on Software Reliability Engineering (ISSRE)*, Trollhattan, Sweden, 2007.

[51] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[52] Liuying Li and Zhichang Qi. Test Selection from UML Statecharts. In *TOOLS 1999: 31st International Conference on Technology of Object-Oriented Languages and Systems*, pages 273–281, Nanjing, China, 1999. IEEE Computer Society.

[53] Xiaojun Liu, Jie Liu, Johan Eker, and Edward A. Lee. *Heterogeneous Modeling and Design of Control Systems*, chapter 7, pages 105–122. Wiley-IEEE Press, New York , USA, April 2003. Chapter in Software-Enabled Control: Information Technology for Dynamical Systems.

[54] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, London, UK, 2002. Springer-Verlag.

[55] Arilo Claudio Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme Horta Travassos. Characterization of Model-based Software Testing Approaches. Technical Report ES-713/07, PESC-COPPE/UFRJ, 2007.

[56] Graeme I. Parkin. Vienna Development Method Specification Language (VDM-SL). *Comput. Stand. Interfaces*, 16(5–6):527–530, 1994.

[57] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[58] Dorina C. Petriu, Christiane Shousha, and Anant Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *Software Engineering*, 26(11):1049–1065, 2000.

[59] R. Pooley. Using UML to derive stochastic process algebra models. In *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW)*, pages 23–33, 1999.

[60] Wolfgang Prenninger and Alexander Pretschner. Abstractions for Model-Based Testing. *Electronic Notes in Theoretical Computer Science*, 116:59–71, 2005.

[61] S. J. Prowell. JUMBL: A Tool for Model-Based Statistical Testing. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 337.3, Washington, DC, USA, 2003. IEEE Computer Society.

[62] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[63] Harry Robinson. Using Pre-Oracled Data in Model-Based Testing. White paper originally appeared internally at Microsoft, 1999. Available at: http://www.geocities.com/harry_robinson_testing/pre-oracled.htm.

[64] Harry Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. In *Proc. 1st European Conference on Model Driven Software Engineering*, pages 118–127, December 2003.

[65] S. Rosaria and Harry Robinson. Applying models in your testing process. *Information & Software Technology*, 42(12):815–824, 2000.

[66] Kirk Sayre and Jesse Poore. A reliability estimator for model based software testing. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, page 53, Washington, DC, USA, 2002. IEEE Computer Society.

[67] Kirk Sayre and Jesse H. Poore. Automated Testing of Generic Computational Science Libraries. In *HICSS*, page 277. IEEE Computer Society, 2007.

[68] D. Seifert, S. Helke, and T. Santen. Conformance testing for statecharts. Technical Report 2003/1, Technical University of Berlin, 2003.

[69] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[70] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, 2nd edition, 1992.

[71] W. Thomas Swain and Stephen L. Scott. Model-Based Statistical Testing of a Cluster Utility. In *International Conference on Computational Science (1)*, pages 443–450, 2005.

[72] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[73] G. J. Tretmans and H. Brinksma. Côte de Resyste – Automated Model Based Testing. In M. Schweizer, editor, *Proceedings of the 3rd PROGRESS workshop on Embedded Systems, Veldhoven, The Netherlands*, pages 246–255, Utrecht, 2002. STW Technology Foundation.

[74] Jan Tretmans. Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, 1996.

[75] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[76] S. Uchitel, J. Magee, and J. Kramer. LTSA-MSC Analyser Implied Scenario Detection. Available at:http://www.doc.ic.ac.uk/ su2/Synthesis/.

[77] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The Universiy of Waikato(New Zealand), 2006.

[78] David White. Distributed systems security. *DBMS*, 10(12):44–ff., 1997.

[79] James A. Whittaker. Stochastic software testing. *Ann. Softw. Eng.*, 4:115–131, 1997.

[80] James A. Whittaker and Michael G. Thomason. A markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.

[81] Guido Wimmel and Jan Jürjens. Specification-Based Test Generation for Security-Critical Systems Using Mutations. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 471–482, London, UK, 2002. Springer-Verlag.

[82] Li Ye. Model-Based Testing Approach for Web Applications. Master's thesis, University of Tampere, Department of Computer Science, June 2007.