Building a Tool for Testing Real Time Systems

By: Manar Abu Talib

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science
at Concordia University
Montreal, Quebec, Canada

March 2004

Your file Votre référence
ISBN: 0-612-90985-9
Our file Notre référence
ISBN: 0-612-90985-9

ABSTARCT

Building a Tool for Testing the Real Time Systems

Manar Abu Talib

Real time systems can be defined as systems whose behavior is time dependent. They don't depend only on the values of input and output signals, but also on their time of occurrence. These systems are highly used nowadays, such as in patient monitoring systems, air traffic control systems, and telecommunication systems.

Ensuring the correctness of real time systems before the development and ensuring that it functions correctly within the specified time constraints become a very important task to avoid catastrophic consequences. A number of design issues affect the testing strategies and the testability of the system. In this thesis, we have designed and built a tool that generates the timed test cases by implementing the state characterization technique. The tool consists of the following steps: First, the real-time system is modeled as a Timed Input Output Automaton (TIOA). Its description is written in a file to be parsed by the tool using JAVACC. TIOA has locations, clocks and transitions with time constraints. Second, the tool samples the stored TIOA into sub automaton easily testable, called Grid Automaton. Third, the tool is then responsible for transforming this Grid Automaton into a Non-deterministic Timed Finite State Machine (NFSM). Finally, test cases are generated from NFSM, using the Generalized Wp-method. We have applied many examples to the tool and it demonstrates its ability to detect many possible faults.

Acknowledgements

The writing of a master thesis should bring feelings of fulfillment, accomplishment and joy. This is surely one of those cases, but it would have been impossible without the help and support of many.

On an academic level, I would like to thank Prof. Rachida Dssouli and Dr. Abdeslam En-Nouaary for their keen constructive criticisms that have guided me in shaping this thesis into its final form as well as for the support and "pats on my back".

On a professional level, I would like to thank all my colleagues at Electrical and Computer Engineering department as well as its staff for their views on the importance of a higher education.

On a personal note I would like to thank the family support from my parents, brothers and sister as well as the patience of my husband Adel in bearing with me throughout the writing process. I would also like to express my thanks to my daughter for her lack of complaint and comprehension.

Table of Contents

List of Figures

List of Tables

CHAPTER I: INTRODUCTION

Although testing principles developed for non-real-time systems are applicable for real-time systems, the fact that time is a parameter in the testing complicates many issues [1]. However, before introducing the testing methods for real time systems, what does real time systems mean? What are their characteristics? What does testing mean? Why is testing important? How to test real time systems? In this chapter, we try to answer the above questions. The objective and the outline of this thesis are also introduced.

1.1 Definition of Real Time Systems

Most of the processors produced today are used in the embedded systems [2], [1]. Many of these embedded systems have real-time requirements and therefore are called real time systems (e.g., airplanes, patient monitoring systems, microwave ovens, mobile telephones and toys). This means that the successful operation in such systems is measured not only on the correctness of the result produced but also on the time of response. Real-time systems, therefore, interact with their environment using time constrained input/output signals. Another definition of a real-time system can be in terms of a "real-time task" where a real time system contains at least one real-time task [1], [3]. In theory, a real-time task is a task that must be completed at specific time (not before or after that specific time) [1], [4]. However, in practice, it is usually enough if the real-time task is completed before the specific time, that is, the deadline.

Real-time systems are often classified according to the cost of missing a deadline. Based on that, Locke describes four different classes (soft, firm, hard essential, and

hard critical) of real-time systems. "In a soft real-time system, completing a task after its deadline will still be beneficial even if the gain is not as big as if the task had been completed within the deadline and it may be acceptable to occasionally miss a deadline. In a firm real-time system, completing a task after its deadline will neither give any benefits nor incur any costs. In a hard essential real-time system, a bounded cost will be the result of missing a deadline. An example may be lost revenues in a billing system. Finally, missing a deadline of a hard critical system will have disastrous results, for instance loss of human lives" [1].

It is to be noted that different inputs to the real time systems lead to input types. If two inputs have entered the same channel in a system and they only differ in the time of entrance, then they have the same input type [1]. Input types may be very different from each other. Temperature readings, keyboard commands and pushing a call button of an elevator are examples of input types. An input type may be periodic, sporadic or aperiodic. If an event occurs with a regular period, then it is periodic. If inputs occur any time and there is a known minimum inter-arrival time between two consecutive inputs then it is sporadic. Finally, if nothing is known about how often inputs may occur or when it is known that inputs may occur anytime, then it is aperiodic.

Real time systems have many characteristics [5]. They are large and complex. They contain many components with complicated relationships between them. However, they have concurrent control of the system components. They also have facilities for the hardware control. Real time systems should be extremely reliable and safe. They have the ability to fail in such a way to preserve as much capability and data as possible and to detect and correct the failure. A processor failure in a non real time

system may result in reduced level of service; however, in a real time system it may be catastrophic such as life and death, financial loss and equipment damage. For example, when a pilot moves his airplane, the computer-controlled system is expected to change the rudder almost immediately. If the rudder is moved too slowly or too fast, the plane might become instable and crash. Still, to the best of our knowledge, little attention has been devoted to develop an automatic tool for testing real-time systems since the time parameter complicates many issues in the testing [1].

To that end, real time systems can be defined as systems whose behavior is time dependent. These systems are highly used nowadays and software engineering is responsible for controlling safety for these critical systems.

1.2 Verification and Testing

Verification and testing are two essential techniques that are used to cope with the correctness of a system. Verification can guarantee the correctness of the system specification. It aims to ensure that the designed specification satisfies predefined functional and timing requirements [6]. Our concern lies on testing, , that is, dynamic execution of test cases [1], [7]. It is an important activity that aims to ensure the quality of the implementation, which verification doesn't guarantee. Testing techniques check whether or not an implementation conforms to its specification [6].

Testing procedure consists of generating test cases and applying them to the implementation, which is referred to as Implementation Under Test (IUT). Three main testing strategies are available, which are white-box testing, black box testing and gray-box testing. Fault model is also another important aspect in the testing of an implementation for a system. It is referred to the potential basic fault that can exist in

an implementation [8] [9] [10] [11] [12]. In fact, some techniques, which generate the test cases, may able to detect all the potential faults, while others may fail in detecting some faults.

Assessing and increasing reliability are the major goals of testing [1], [13]. Based on monitoring the number of encountered failures, the test cases are chosen and executed to assess the reliability. A failure is defined as "a deviation of the software from its expected delivery or service" [1], [7]. To increase the reliability, it depends on selecting test cases that are assumed to be especially likely to detect failures. The observed traces are analyzed to find the cause of the failure, which is the fault [7]. By removing the fault, the reliability is assumed to increase. The power of test cases generation technique to detect faults in an implementation is referred to as fault coverage. Many different test methods exist (e.g. formal methods based on FSM and EFSM [14]) that are all assumed to generate test suites containing test cases especially prone to revealing failures. These test methods can be compared according to their respective fault coverage. One method is more powerful than the other if it has a better fault coverage. In this research, Timed Wp-Method is chosen for the derivation of test cases for real-time systems modeled as a TIOA, a variant of the Alur and Dill timed automat a model [15] [16].

Gray-box testing is applied as efficient strategy to test real time system using Timed Wp-Method. Before explaining that strategy we have to indicate that in white- box testing, the test suite is generated from the implemented structures, while in black-box testing the structure of the implementation is not known and the test cases are generated and executed from the specification of the required functionality at defined interfaces. In gray-box testing, the modular structure of the implementation is known

but not the details of the programs within each component. As we can see in Figure 1, the testable model consists of two parts: the control part and the clock part.

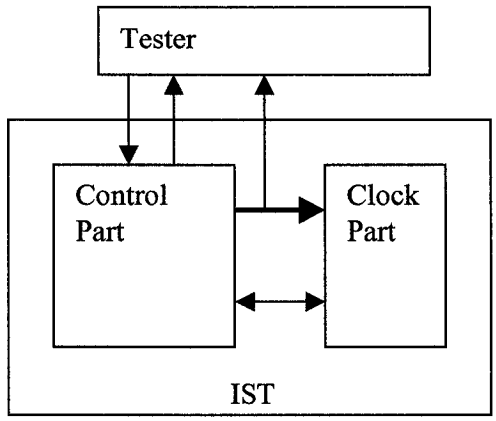

Figure 1: Test architecture

The tester knows and reaches some of these parts where he can observe when the clock resets to zero. The control part indicates the interaction between the timed system and its environment by exchanging input and output signals. However, the clock part handles the clock variable, used in the specification of the timed system. There are two channels, which are connecting the two parts. One channel is used to exchange some internal signals after the control part receives an input from the environment and checks with clock part whether or not the input satisfies the time constraint of the specification. In other channel, clock reset signal is sent by control part to the clock part asking it to reset the clock to zero.

1.3 The Objective of the Thesis

Based on the work published by Dr. Abdeslam En-Nouaary in [34], this thesis is intended to develop a tool that generates the test cases for testing real time systems

using the Generalized Wp method (Figure 2). The developed methodology is composed of five steps:

1. Sampling the Timed Input Output Automaton (TIOA) by using a suitable granularity, in order to have a Grid Automaton, which is a simplified sub automaton that can be easily tested [6].

2. The transformation of the Grid Automaton into a Non-deterministic Timed Finite State Machine (NFSM) [6].

3. The minimization of the obtained NFSM.

4. The generation of test cases based on the generalized WP-method [6].

5. The optimization of the obtained test cases. The fault coverage of test cases generation method is then estimated and its ability is shown to detect many possible faults

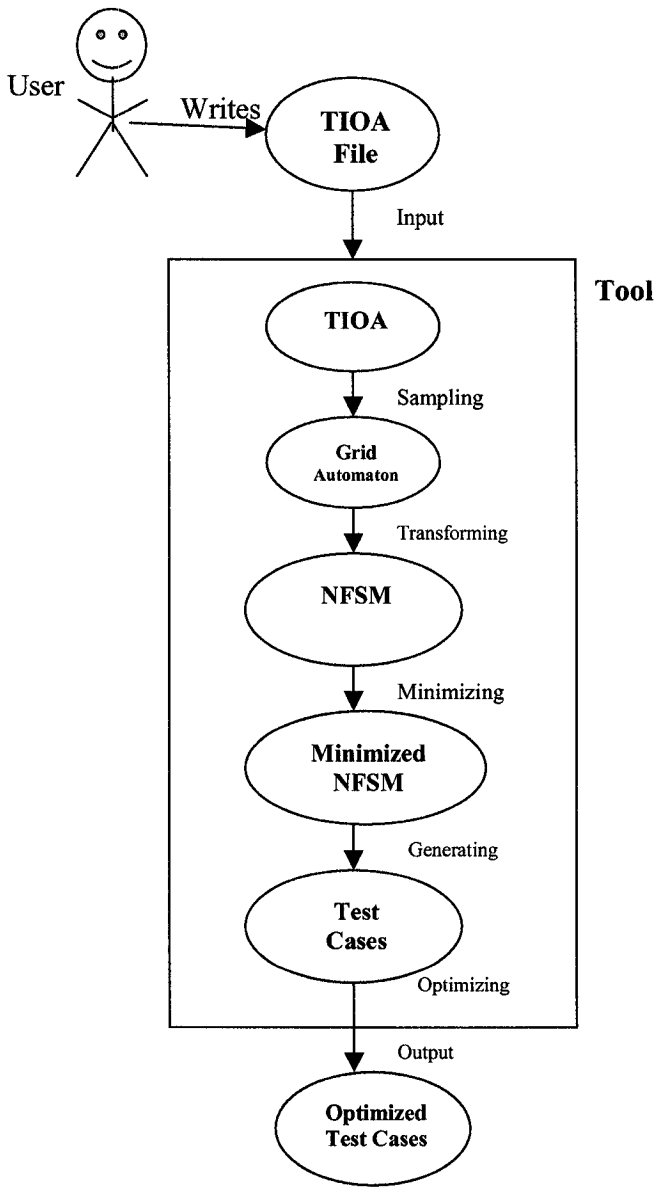

Figure 2: The context diagram of the developed tool

In this thesis, we have designed the tool and chosen the appropriate algorithms and data structures. We have chosen JAVA language to implement our tool for reasons listed in Chapter 3.

1.4 Outline of the Thesis

The rest of this thesis is structured as follows: **Chapter 2** provides an overview of some model based testing methods. In order to have a well-designed tool, **Chapter 3** explores the data structures and the algorithms. **Chapter 4** includes the implementation details and the solutions of the major design decisions and the design quality attributes are introduced. **In Chapter 5**, many examples are examined and evaluated using the tool. We also validate our results at the end of that chapter. Finally, **Chapter 6** provides a conclusion of what was achieved and why the work is important. In addition, pointers for future research are given.

CHAPTER II: STATE OF THE ART

The testing phase represents a large effort within the common software development cycle. Generation of the test cases is the first step in testing the software process. The most difficult problems in generating the test cases are finding a test data selection strategy that is both valid and reliable [17]. In this chapter, we present the state of the art of the test cases generation methods.

2.1 Testing Based on FSM Model

The Finite State Machine (FSM) model [18] is the base of the most research contributions in test derivation and selection. It was used since 50s in testing the sequential circuits, the communication protocols and the object-oriented systems. One raised problem with the test based on FSM Model is the conformance testing. We want to test whether the implementation Mi conforms to the specification Ms through observing the Input/Output behavior of Ms. To do so, we define a conformance relation to link Mi with Ms. The most conformance relation used is the traces equivalence. It is based on the observable behavior of the implementation after the application of the test suites.

The test cases generation for FSM is often done under certain hypotheses of Ms specification and the fault types that can be presented in Mi implementation. The hypotheses of Ms specification are used in order to ensure that the specifications with the desired properties are treated. The implementations' hypotheses and the concerned possible fault types are used to limit the implementation's domain that will be considered. Without these hypotheses, each FSM can be considered as a possible

implementation of a given specification, thus the number of implementation machines will be infinite.

In general, the specification is supposed to be an FSM that is:

- Deterministic: where δ: $S \times I \rightarrow S$ is a transfer function, λ: $S \times I \rightarrow O$ is an output function, S is a state, I is an input and O is an output.
- Reduced: there are no equivalent states in FSM. Refer to 3.5 section.
- Strongly connected: any state can be reached from the initial state.
- Completely specified: where δ: $S \times I \rightarrow S$ and λ: $S \times I \rightarrow O$ are complete functions that reach to any input. Empty output is also allowed; but the completeness is implicit.

However, the implementation is often supposed to be an FSM that is:

- Having a limited number of extra states.
- Deterministic.
- Strongly connected.
- Completely defined: providing an answer for any input in a limited time.
- Implementing correctly the reset action.

Many of these assumptions can be avoided and methods have been developed for partially specified and nondeterministic behaviors (see the following sections). Now with the above hypotheses, the FSM fault model can be summarized with the following four fault types [19]:

- Output faults: these faults cause a state not to respond with an expected output.
- Transfer faults: these faults cause a state with a specific input or output to enter a state different from the expected one.

- Transfer faults with the additional states: the arrival state of a transition is a new one.

- Transitions' faults (additional or missing transitions): usually, we suppose that the automaton is deterministic and completely specified, which means that for each input of a state, there's exactly one corresponding transition coming out of that state. The absence of this transition is also a fault of this type.

Many methods of test cases generation from FSM were developed. There are two categories: methods based on states' identification and others based on simple path of all transitions. In the following sections, we discuss some of these methods.

2.1.1 Transition Tour Method

The transition tour method [20] consists of generating a single input sequence that traverses all the transitions at least once and returns to the initial state. Obviously, the generated sequence can contain many redundant inputs that generate cycles in the sequence. This can be avoided by optimizing the transition tour. Some efficient algorithms such as Chinese Postman Algorithm for the transition tour derivation of certain FSM classes were studied in [21].

The problem of finding the optimal transitions in FSM is a well-known problem in the graph theory. In fact, FSM is seen as oriented graph and optimal transition tour, which seems exactly like Euler tour. An Euler tour in an oriented graph is the transitions' sequence, which starts and ends at the same state and it contains each transition at least once. To have an Euler tour in an oriented graph, the graph has to be:

- Strongly connected.

- Symmetric: a graph is called symmetric if and only if the number of the incoming transitions for each state in the graph is equal to the number of the outgoing transitions for the same state. A non-symmetric graph can be

transformed into a symmetric one by an incremental procedure [22]. This consists of duplicating certain transitions till the graph becomes symmetric.

The fault coverage of the generated test cases by the method of transition tour is not complete. In fact, the range of test cases checks only the transitions' outputs. This is not enough since it has to check the state where the implementation under test exists after each transition. This method has a limited error detection power compared to other methods since it doesn't consider the state verification. However, an advantage of this method is that the test sequences obtained are usually shorter than test sequences generated by the other methods.

In Figure 3, the possible transition tour is formed by the transitions' sequence: t_1. t_4. t_3. t_6. t_7. t_8. t_2. No transition needs to be traversed twice. It's clear that the arrival state of the last transition, t_2, is not checked. As a result, if t_8 transition reaches to *S2* instead of *S1*, this transfer fault will not be detected. In order to systematically detect the transfer faults, one has to identify the state into which the implementation goes after the execution of the tested transition.

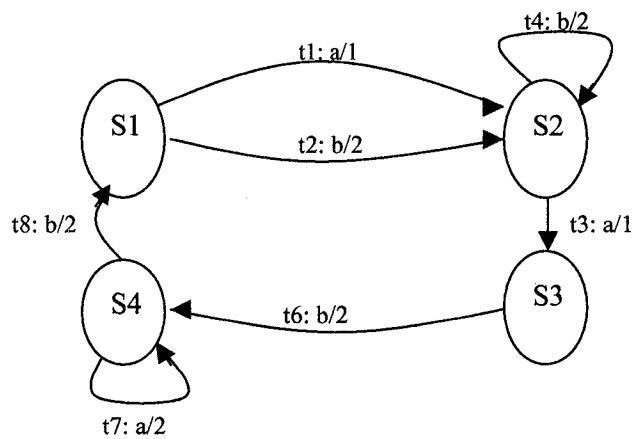

Figure 3: FSM example (1)

2.1.2 Distinguishing Sequence Method

The distinguishing sequence (DS) method [23] is a state identification sequence and it is applicable if FSM has one distinguishing sequence. A distinguishing sequence is a sequence of inputs that produces a different sequence of outputs for each FSM state: sequence x that produces different output for each state such that for all pairs t, s with $t \neq s$: $\lambda(s, x) \neq \lambda(t, x)$.

The DS method consists of two steps: the state verification and the transition verification. During the first step in the state verification, the implementation has to provide the observed output to the distinguishing sequence for each state. For example, by applying the sequence of inputs in the current state of FSM such that from the outputs we can identify whether or not that state is where we should go. The second step of the DS method consists of checking each FSM transition.

The states' verification is done by applying the following procedure to each FSM *si* state:

- The implementation is carried out at i_i state, which is supposed to be isomorphic to s_i state, by applying the reset action at the initial state followed by the *preamble* from the initial state s_0 to s_i state.
- The DS sequence is submitted to the implementation to check whether or not the observed output is equal to the excepted output.

This phase ensures that the implementation has at least n states $i_1, i_2, i_3, \ldots, i_n$, which are accessible by the same preambles of their isomorphic states $s_1, s_2, s_3, \ldots, s_n$ in the specification.

However, many transitions are used in the distinguishing sequence preamble during the states' verification. These transitions must be implemented correctly. It's the

transitions' verification step that has the role to test each transition s_i *to* s_j according to the following procedure:

- The implementation is carried out at i_i, which is isomorphic to s_i, by applying the reset action at the initial state followed by the preamble *Preamble* (s_i);
- The i input is submitted to the implementation and the output is observed to check if it is the same as the expected output;
- By the application of DS sequence, the new implementation state is tested to check if it is i_j.

The DS method guarantees complete fault coverage (detecting both transfer and output faults). However, the disadvantages of this method is that a DS may not be found for a given. Also applying a fixed length sequence may not lead to the shortest state identification sequence. Table 1 shows that DS = a.a for the FSM shown in Figure 4. The corresponding outputs sequences and the preambles for each state are also shown in the table.

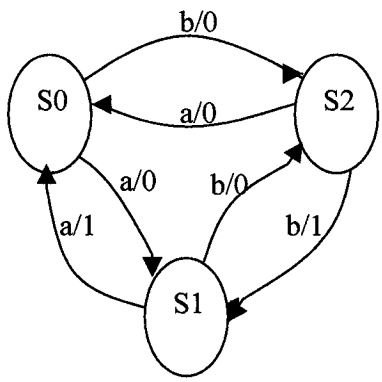

Figure 4: FSM example (2)

Table 1: DS of Figure 4

State	Output for DS = a.a	Preamble
S0	0.1	Null
S1	1.0	a/0
S2	0.0	b/0

2.1.3 UIO Method

The unique Input/Output sequence (UIO) method consists of generating test sequences that check if the arrival state of each transition is the right one (or the transfer is correct). To do so, it uses the unique Input/Output sequence of the arrival state. It allows distinguishing this state from others.

To apply the UIO method, the FSM specification has to contain an initial state. As in DS method, each state of this FSM must have its own Input/Output sequence. The test procedure done for each transition s_i to s_j is as follows:

- Bring the FSM to its initial state, s_0, using the reset operation.
- Find the shortest path from the initial state to s_i.
- Apply i input to do the transition execution and check if the output is as expected.
- Apply the unique Input/Output sequence of s_j to check if it's the good state.

It happens sometimes, that the unique Input/Output sequence doesn't exist for a state. In this case, the signature technique is used. The signature of s_i is a sequence formed of a set of minimal Input/Output sequences, IO(s_i, s_j), each one of them starts at s_i and distinguishes s_i from another state s_j ($j \neq i$).

In the original paper where the UIO method was proposed, the authors used only the verification of transitions to generate the range of the test cases because they thought that the transition verification is enough to guarantee good fault coverage. In [24], Vuong took this lack into consideration by showing certain faults that can be hidden from the tester. As a result, he recommended adding the states' verification step to the UIO method, which gives a new method, called UIOv method. This step of states'

verification consists of ensuring that the Input/Output sequences are unique in the implementation under test. To do so, the UIO (s_i) sequence for s_i state in the specification is accepted by its isomorphic state in the implementation, and it's rejected for the rest of states. The reject procedure of UIO (s_i) sequence by a state s_j is the following:

- Bring the implementation at i_j state, which is supposed to be isomorphic to s_j state, by applying the reset action in the initial state followed by the preamble from the initial state s_0 to s_j state.
- Submit the UIO (s_i) sequence to the implementation and check if the observed output is really different from the expected one by applying the UIO (s_i) to s_j.

In Figure 5 and Table 2, we present an FSM and its corresponding UIOs for each state.

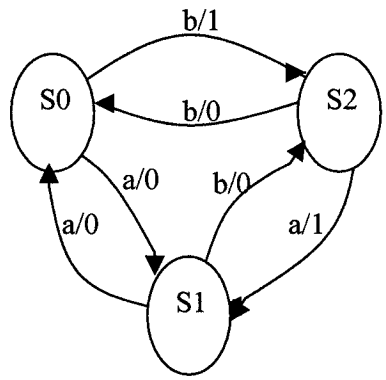

Figure 5: FSM example (3)

Table 2: UIO sequences of Figure 5

State	UIO (s_i)	*Preamble* (s_i)
S0	b/1	Null
S1	a/0.b/1	a/0
S2	a/1	b/1

2.1.4 W Method

The DS method has the advantage of having complete fault coverage. However, its major disadvantage is that the DS sequence doesn't exist for all FSM. This limits the application of DS method and constitutes a kind of motivation to look for more general method. The W method [17] is a general method, which is applicable for all minimal FSM. It's based on the fact that for all FSM, there exists a set of finite input sequences, called W characterization set, such that the output sequences are unique for each state.

Besides W set, W method has a second set of input sequences, called the *set of transitions cover*. We write P for any set of input sequences (including the empty set), which tests for each transition s_i to s_j of the specification. This means, the set P consists of all partial paths and it can be formed from a testing tree, which shows every transition from state s_i to state s_j on each input.

The W method produces a set of test cases, which are formed from the concatenation of Z and P sets (e.g. P.Z) where $Z = (\{\varepsilon\} \cup I \cup I^2 \cup \ldots \cup I^{m-n}) .W$. The use of Z set instead of W set is principally due to the fact that the number of states, m, in the implementation can be greater than the number of states, n, in the specification. If $m = n$, we will have $Z = W$ and accordingly, the range of test cases will be $P.W$. Each test sequence starts with the initial state and returns to it again.

The assumptions about the implementation under test are that the implementation is minimal, may start in a fixed initial state, and is strongly connected. Under these assumptions a W-set exists, and the W-method gives a set of sequences that are guaranteed to detect any misbehavior of the implementation. However, the disadvantage of using this method is that the W-set may not exist for every FSM,

especially if it is an incompletely specified machine. Therefore, before using this method, if the machine doesn't have a W-set sequence, one should make the machine to be a completely specified. For example, adding an "error" state and declaring all unspecified transitions to lead to this state.

The main goal of the W method is to test whether or not an implementation conforms to FSM specification. To do so, we check for each input sequence whether or not the observed outputs are the same as the expected outputs. The set of tests *(P.Z)* detects all output and transfer faults since the number of states in the implementation doesn't exceed *m*. Figure 6 and Table 3 show respectively a reduced FSM and the corresponding W set.

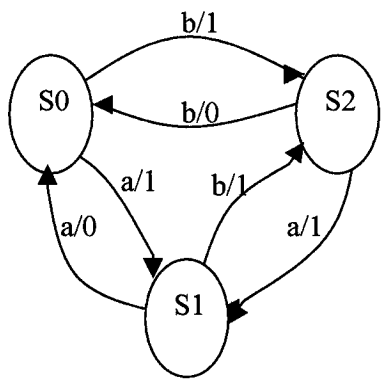

Figure 6: FSM example (4)

Table 3: W set of Figure 6

State	Output for W = {a, b}	Preamble (P)
S0	{1, 1}	Null
S1	{0, 1}	a/1
S2	{1, 0}	b/1

There are two modes to check the verification [17] of test sequence: the test mode and the walkthrough mode. The test mode is to build the correct responses in the form of the operation sequences, which is based on the specification and the semantics of the

operations. The responses from the design are then derived and compared with the correct or expected version. In the walk-through mode, the input sequences and their corresponding responses from the design are represented as "path programs". The correctness of these programs can be established by a walkthrough procedure based on the specification. When the testing sequences are got from the concatenation of P and W, it can be decided whether or not a path program is correct.

The proof of the fault detection power of W-method is given in [17].

2.1.5 The Partial W Method

In [25], the partial W (Wp) method is introduced to determine whether or not a given implementation protocol satisfies the properties required by the specification protocol. The Wp method is a generalization of UIOv method which is always applicable. It is at the same time an optimized and a refined version of W method since it reduces the length and the number of the test cases. Therefore, the assumptions made for Wp method is similar to those made for W method. FSM is strongly connected and deterministic. It also contains no equivalent states and it is a completely specified machine. Instead of using W set to check each reachable state s_i, only a subset of this set can be used in certain cases. These subsets are called the state identification set (Wi) for s_i. *Wi set* is a set of s_i, if and only if, Wi is minimal and for each state s_j $(j \neq i)$, it exists a sequence of inputs which belong to Wi and generates two different behaviors in s_i and s_j. The Union of all Wi produces W characterization.

The Wp method consists of two phases. The first phase checks that all states defined by the specification are identifiable in the implementation, and each s*i* state in the

implementation can be identified by the *Wi* set. The test cases that allow achieving these objectives are given by Q. ({ε} U I U I^2 U ... U I^{m-n}) .*W*, where *W* is the characterization set and, *m* is the maximal number of states in the implementation, and Q is the cover set of states. The Q set consists of the input sequences that allow reaching the specification states from the initial states. At the same time, the transitions leading from the initial state to these states are checked for the correct output and the state transfer.

The second phase checks the implementation of all specified transitions, which aren't tested in the first step. The test sequences of phase 2 consist of the sequences of P which are not contained in Q, concatenated with the corresponding Wi, written R×W, where R=P-Q, that is the set of all transitions that is not part of Q. It is noted that different test sequences are obtained depending on the choices made for the sets P,Q and Wi.

Finally, the test hypotheses of Wp method are the same as W method and its fault coverage is complete. The following table shows *W* and *Wi* sets (i = 0, 1, 2) for FSM in Figure 6 and the corresponding outputs.

Table 4: W and Wi results

State	Output for W = {a, b}	Preamble (s)	W1 (Input)	W1 (Output)
S0	{1, 1}	Null	{a, b}	{1, 1}
S1	{0, 1}	a/1	{a}	{0}
S2	{1, 0}	b/1	{b}	{0}

2.1.6 The Generalized Wp Method

The generalized *Wp* method, generates test sequences for both non-deterministic and deterministic finite state machines. The generalized *Wp*- method [26] is based on extending the state identification approach used for deterministic finite state machines to non-deterministic finite state machines (NFSM). To do so, the NFSM is transformed to an observable NFSM. ONFSM has a property that a state and an input|output pair uniquely determine the next state, while a state and an input alone do not necessarily determine a unique next state and an output.

The generalized *Wp* method uses the same sets as *Wp* method uses but with the fairness hypothesis. This hypothesis means that it's possible to reach all accessible states by one input sequence *t* in one implementation by applying *t* on it for n finite times. Without this hypothesis, no set of test cases can guarantee complete fault coverage for a non-deterministic implementation. Refer to 3.6 section for a detailed example of this method.

The generalized *Wp* method is applicable for concurrent systems that are modeled as FSMs, which communicate with each other through channels: first, the FSMs of the system are composed, by using the accessibility analysis, to obtain one FSM. After that, the generalized *Wp* method is applied on that FSM. Refer to 5.4 section for an example.

2.2 Real Time Systems Testing

The last three decades have known intensive research activity in the un-timed test domain. However, the computer systems are more specified nowadays with the temporal constraints in order to control their execution. This motivated researchers to

develop new test methods for these systems. In the following, we expose a list of researches about generating the timed test cases.

2.2.1 Test Cases Generation for Theory Formulas

This approach [27] consists of generating test cases from a theory formula. The used logic is an extension of the classical timed logic in order to model the real time systems. The time aspect is expressed according to two fundamental constructors, future and past, referring respectively to the moments in the future and the moments in the past. In this theory, one clock is used.

To generate test cases from formula F, F is decomposed in a hierarchical structure in terms of the atomic formulas that contain events only. An event is a couple (L, t) that is formed from L literal and t moment. A set of events that doesn't contain any contradiction and is happening at the same time (the two events (L, t) and $(\neg L, t)$) is called a history. Each leaf, that doesn't contain any of the decomposition tree of a formula F, is considered as F history. A history of the formula F is called "complete" if it contains a unique value of truth for each predicate formula and at any point in the temporal domain. A test case of formula F is a complete history satisfying F in many points of its temporal domain. This domain is discredited in the digital values.

A complete test case of F formula is constructed by the application of the *offset* and the *concatenation* operation of the test cases with the small sizes, called *elementary test cases*. These are directly extracted from F decomposition tree. The offset Δ of the test case time unites, which satisfies F formula at t moment, consists of the test case

satisfying F at $t + \Delta$ moment. Beside that, the concatenation of two test cases TC_1 and TC_2 is the test case TC_3 obtained by the union of TC_1 and TC_2 event sets.

This method of generating the timed test case, suffers from the following lack: the range of the generated test cases can cover only the digital values of the temporal domain for the formula specification logic. The used logic can only serve in describing very restrictive classes of the real time systems (Those which are specified according to one clock).

2.2.2 Test Cases Generation for FSM Model with Timers and Counters

This method [28] generates the timed test cases from a specification given as an FSM with timers and counters by using *Wp* method [25]. First of all, the behavior of each timer and counter is modeled as FSMs. Then, all the FSMs are combined so that they have only one global FSM, which describes the system. Finally, the *Wp* method is applied on the resulting FSM with certain hypothesis.

FSM timer is defined by:

- Two states indicating respectively the activity and the inactivity timers;
- An alphabet, which forms the corresponding events in respect to the triggering timer, stop and expiration, and the flow period in which the timer is activated;
- A set of the transitions between the states with the appropriate events.

FSM counter is defined by:

- A set of the states representing the different counter values;
- Two events representing respectively the reset and the incremental counter;
- A set of the transitions between the states with the appropriate events.

The problem of this approach is being not applicable for all systems such as systems, which have the general shape of the temporal constraint.

2.2.3 Test Cases Generation for Timed Automaton

Springintveld and al. [29] present a theoretical framework for generating the test cases from Timed Input Output Automaton (TIOA). They adapt W method [17] with taking into account the temporal aspect of the real time systems. The conformance relation is used to judge whether or not the implementation conforms to its specification [30].

The approach contains many transformations. First of all, we count the number of TIOA region, which is a result from the automaton that is composed of both the implementation and the specification. This aims destructing the maximum traces length of the states' distinction. Then, we construct a sub-automaton from the region graph, called Grid Automaton, by using a "*uniform mapping*" [31]. This characterizes the relation ~ between the clocks' interpretations. The resulted automaton is finally used to generate the test cases by applying W method.

Grid Automaton is the labeled-transitions system summarizing the region graph behavior, which is necessary for the system exhaustive test. Each state in this automaton has a delay transition. Its period is 2^{-n}, where n is the length of the states' distinction trace in the region graph for the automaton (that is composed of the implementation and the specification). In the worst case, n is equal to the number of the clocks in the region graph. The value 2^{-n} is selected to allow the test cases, under certain hypotheses, detecting all possible faults in the implementation.

The main result of this work is the proof of the possibility for the exhaustive test to detect all faults in the implementation using the conformance relation. However, the major problems of this method are the restriction of the used model and the number of the generated test cases. The used model is TIOA in which the system outputs can exist in integer values of the clocks' temporal space. This restriction doesn't allow describing a large range of the real time systems where their outputs can be produced within the temporal range and at a specific point. For the sequence of the generated tests, its number is so big, even for a simple academic example. In fact, this big number of test cases is due to the transitions' period of 2^{-n} delay used in constructing the Grid Automaton. This period is very small because the number of region clocks in TIOA is exponential to the number of clocks and the constants used for these clocks' constraints.

The authors don't use the exact number of the region clocks, but an approximation number to the superior bound provided by Alur and Dill [32].

2.3 Conclusion

In this chapter, we have reviewed the methods of the test cases generation from the most known formal models. More precisely, we have presented the test based on FSMs and the timed test. For each method, we have explained its strengths, its weakness and the conditions of its application. The Generalized *Wp* method will be used in next chapters to be applied in our developed methodology.

In the following chapter, we deeply discuss our developed methodology and its structure and processes. We propose the algorithms and the reasons of using them.

CHAPTER III: TIMED TEST CASES GENERATION

For any tool to be built, it is an important task to go through its structure, functions and goals before designing it. Chapter 3 is based on IEEE standard [33] to point at these issues in order to not miss any small detail. Although it doesn't specify any particular design, it has the advantage of limiting the range of valid design. After having the clear picture of what the tool is, its functions and its goals, the areas of data structures and algorithms are introduced in this chapter. The next aim is not only designing the correct tool but it is more about evaluating the tool design in order to have quality attributes in it. As a result of this chapter, the implementation of the tool is as easy as mapping the data structures and algorithms to the direct code. Figures and examples are represented for understanding the steps of our developed methodology.

3.1 Tool Input, Process and Output

The tool should support the Generalized Wp method. The testing of the real time system should be done using the Generalized Wp method.

The steps of the test process are:

- The tool receives the Timed Input Output Automaton specification (TIOA) from the tester.
- The tool displays its Grid Automaton information, its NFSM information, its test suites and the detailed information about them such as Q set, R set...etc.
- The tester validates the results by using oracle (see section 5.5) to compare the observed outputs with the expected outputs.

The following figure presents an outside view of the tool. The tool has one primary actor (the tester) who initiates the activity by describing the real time system in terms of TIOA. Therefore, the input of the tool is TIOA specification. The tester is also responsible of running the tool to derive the test cases from the given TIOA based on timing constraints of the system. Therefore, the test suites, the Grid Automaton details and the Minimized NFSM details are the output of the tool.

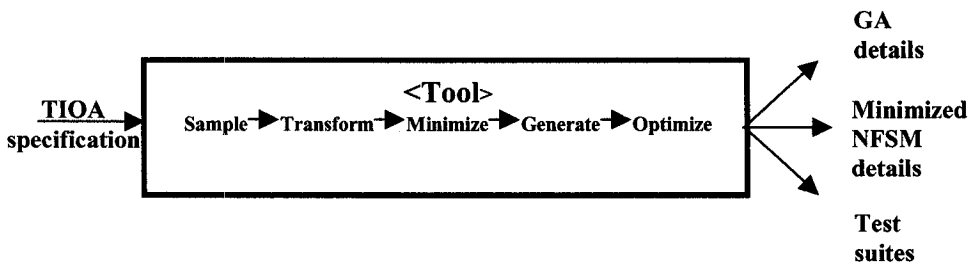

Figure 7: The tool input, process and output

As we have noticed, the tool needs sufficient memory to store the information related to TIOA, Grid Automaton, NFSM and to run the tool in order to generate the test cases.

In the next sections, we explained in details the steps of our developed methodology, why we need each step, what the selected algorithms are and their time complexity.

3.2 TIOA Specification

TIOA specification is a file, which needs to be parsed by the tool in order to generate its timed test suites (test cases). A set of locations, a set of inputs, a set of outputs, a set of clocks and data variables and a set of transitions define TIOA. This file must follow a specific format (Figure 8).

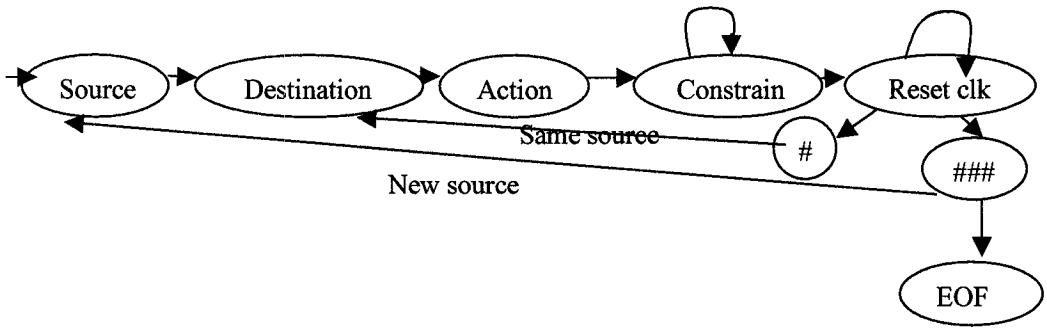

Figure 8: File format

The tester starts describing TIOA by going to each location and writes its name (source name), its destination, its action, its constraints and its reset clocks information. One location may have more than one transition, and the tester has to put "#" sign, which indicates that the next transition is following. After specifying the information of the first location, the tester put "###" sign to start the next location in the same manner as explained before by describing the elements of its transitions. Finally, when the tester finishes all the locations, s/he writes EOF, which means the end of the file. For example, the file format of Figure 9 is:

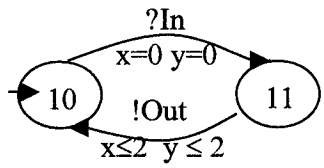

Figure 9: TIOA specification

Source: l0
Destination: l1
Action: ?ln
Constraint : null
Reset clk : x , y
###
Source: l1
Destination: l0
Action: !Out
Constraint: $x \leq 2, y \leq 2$
Reset clk: null
###
EOF

3.3 Sampling the TIOA

Sampling TIOA into Grid Automaton is the first step of our developed methodology. TIOA specification is the input of this step and Grid Automaton is the output. We need such step since Grid Automaton is sub-automaton that is easily transformed into NFSM.

The Grid Automaton is to represent each clock region with a finite set of clock valuation. This finite set is called "representatives of the clock region" and it is determined from the set of the grid points with the granularity at most 1/ (1+n) where n is the number of the clocks. The granularity is determined according to the number of the clocks. By this granularity, the clocks are authorized to pass from one clock region to another one, and the automaton is allowed to make a transition from one location to another [6].

3.3.1 Sampling Process

To construct the Grid Automaton of a TIOA, the following operations have to be proceeded [6]: first step, determine the maximal granularity we can use for a specific TIOA. If the number of the clocks is one then the granularity is ½, but if the number of the clocks is greater than one then the granularity is 1/ (n+2) where n is the number of the clocks. By this granularity, the relationships between clocks are kept because every delay transition in the region graph is matched with a transition on 1/(n+2) in the Grid Automaton. Therefore, there will be no relationship between clocks when the number of clocks is one. See Figure 10 as an example of a TIOA Specification, which has one clock "x", then the granularity is ½. Second step, create the initial state of the Grid Automaton formed from the initial location of TIOA and a valuation that sets all

clocks to zero. The Initial state of TIOA specification in Figure 11 is "100", which will be the initial grid state that has the valuation of "x" to be equal to zero. Third step, Create all states reachable from the initial state with repetitive 1/ (n+2) (or ½ if number of clocks is one) delay transition. (100, 0) has a reachable state (100, ½) and the delay transition ½ (See Figure 11). Fourth step, for each state, we create all its possible transitions with a condition that for each transition in TIOA, the clock valuation satisfies the clock guard in the Grid Automaton. We noticed that (100, 0) has two transitions. One transition goes to (101, 0) and x to be reset to zero and the second transition goes to (111, 0) since the clock valuation of (100, 0), which is zero, satisfies the clock guard (zero < 1). Finally, we repeat the process starting with the next state. In our case, (101, 0), (111, 0) or (100, ½) is the next state. The final Grid Automaton of Figure 10 is shown in Figure 11. We noticed that the state (100, ∞) will not have an input transition that goes to (111, 0) since the bound of the time constraint is $x \leq 1$. Besides, its delay transition has a destination (100, ∞) where ∞ means infinity. Note that the example is for understanding the concept; it may not represent the good real-time system.

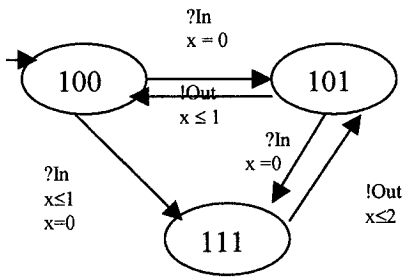

Figure 10: TIOA example with one clock

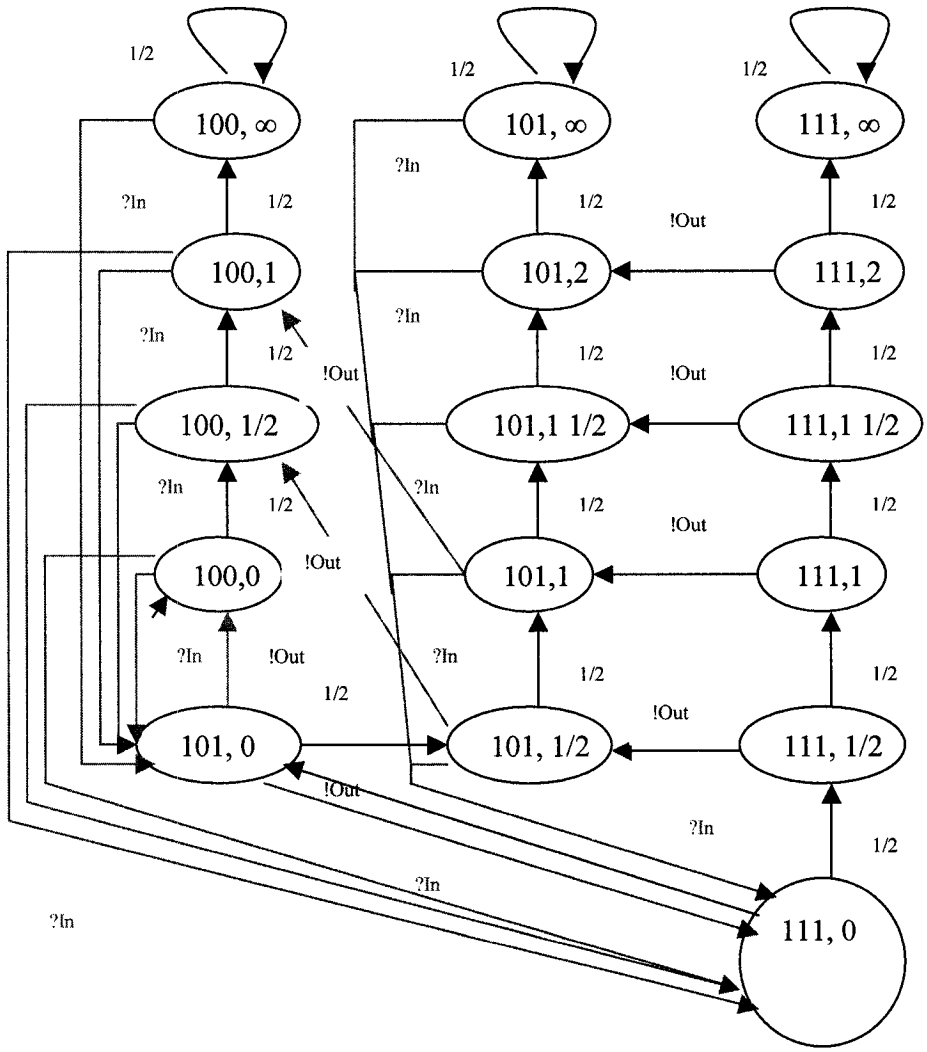

Figure 11: Grid Automaton of Figure 10

3.3.2 Sampling Algorithm

As we have explained the sampling process in the previous section, the corresponding algorithm is stated in this section. The time complexity of the sampling algorithm is: O(C+ S* t) where C is the number of clocks, S is the number of states and t is the number of transitions.

Input: TIOA_States object
Output: Grid_States object
Initially:

- reachable_states and handled_states are empty and their types are Grid_States

- clk_no is an integer variable that has the number of clocks of a TIOA
- granularity is a double variable that has the granularity of Grid Automaton

```
BEGIN
//find the number of the clocks of the TIOA
clk_no = find_clk_no(TIOA_States);

//compute the granularity of Grid Automaton
Granularity = compute_granularity(clk_no)

Create the initial state of Grid Automaton which has the same location as TIOA initial state
Create its Clk Vector object whose size is equal to clk_no

For each Clk object
 {
   Give the clock name   //from the TIOA_States information
   Give value of 0
  } End For

//Add that state g_state to reachable_states
reachable_states.gstate.addElement(g_state);

While (reachable_states - handled_states is not empty)
{
  Get a state s from (reachable_states-handled_states)
  Add s to handled_states

      For (each transition belongs to TIOA transitions)
       {
            If (the clock valuation of s satisfies the clock guard || there is clock reset)
            && (the s location is equal to the source location of that TIOA transition)
            {
                  Create new Grid transition which has the same information as TIOA transition
                  Add the transition to the Grid Automaton transitions of s without including constraints and
                  clock reset information
        } End If

       If (it doesn't exit)
            {
       Add the destination of the transition to reachable_states if it doesn't exist in it
        } End If
       } End For

If (the delay transition exceeds the bound of the maximum time constraint)
          {
         Create delay transition, which has destination with a clock value of infinity
            If (if it doesn't exist in Grid Automaton transitions of s)
                  {
                  Add this delay transition of that state s to its Grid Automaton transitions
            } End If
      } End If

Else
          {
             Create delay transition, which has destination with a (clock value + Granularity)
            If (if it doesn't exist in Grid Automaton transitions of s)
                  {
                  Add this delay transition of that state s to its Grid Automaton transitions
            } End If

      } End Else

If (destination of the delay transition is not included in reachable_state)
          {
         Add destination of the delay transition to reachable_states
      } End If
} End While
Return reachable_states
END
```

Figure 12: Sampling algorithm

3.4 Transforming of Grid Automaton into NFSM

This is the second step of the developed methodology, which takes the Grid Automaton information as input and produces a NFSM as output.

3.4.1 Transformation Process

The transformation of the Grid Automaton into a NFSM is based on Figure 13 [6]:

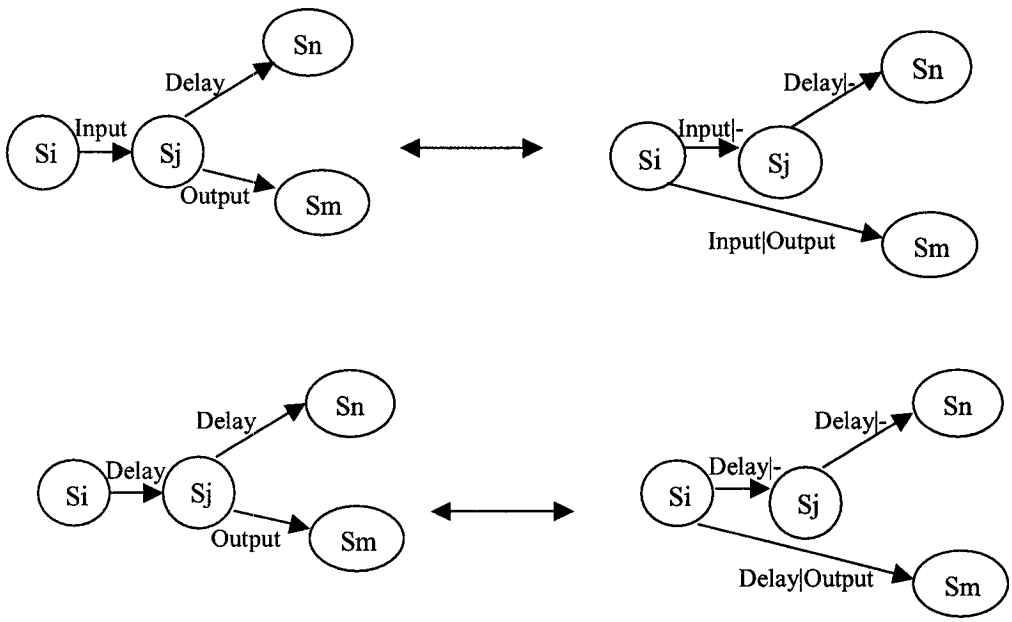

Figure 13: NFSM transformation

The transformation can be explained easily from the above figures. It consists of combining each output with the input or each output with the delay, which is preceding it in the Grid Automaton [6]. Figure 14 shows NFSM of Figure 11. For example, in Figure 11, the Grid Automaton has (100, 0), which has the transition that is going to (111,0) with the input action ?In and (111,0) has the transition that is going to (101,0) with the output action !Out and the clock reset action. The resulted NFSM is a state (100, 0), which has ?In|!Out&Reset action to the destination (101,0). The

transitions with ?In|- action are not shown in the figure to not make it so complicated, however we have to include them in our tool. For instance, (100, 0), (100, ½), (100, 1) and (100, ∞) have such transition, which is going to (101, 0).

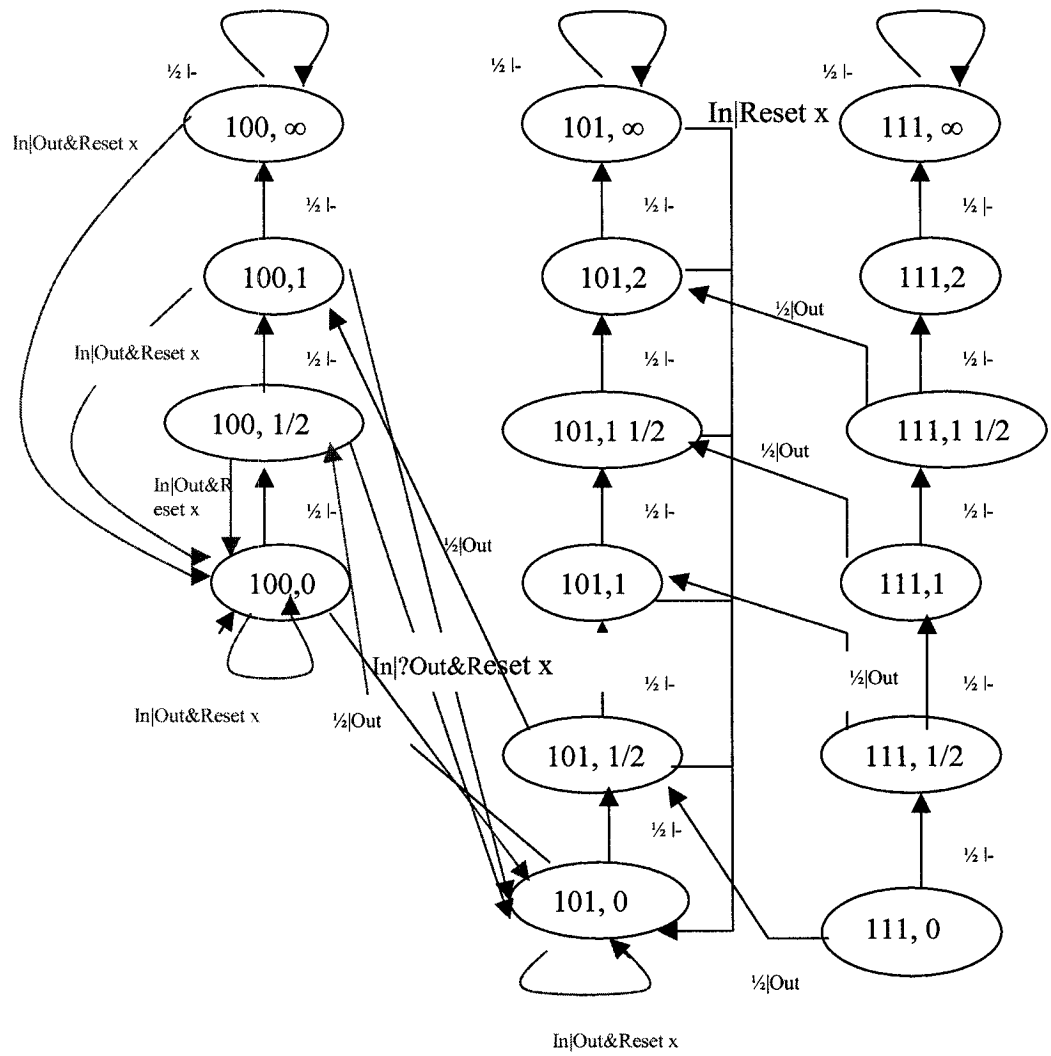

Figure 14: NFSM of Figure 11

3.4.2 Transformation Algorithm

The corresponding algorithm of the transformation process is introduced in this section. The time complexity of the transformation algorithm is: O(S* t*t) when t> I

and t> C and it is O(S*t*C) when C>I and C>t, however, it is O(I*S*t) when I>C and I>t where C is the number of clocks, S is the number of states, t is the number of transitions and I is the number of input list elements.

You can see that the last part of the transformation algorithm makes NFSM to be completely specified. To do so, we have to check each state in the produced NFSM if it contains each input. If not, a transition with that input action is added in that state. The transition is going to itself without any output action.

Input: Grid_States object
Output: NFSM_States object
Initially: NFSMstate is an empty vector of Location objects

```
BEGIN
While not the end of Grid Automaton states
  {
    Get a Grid Automaton state s
    Make a new NFSM state s
    While not the end of transitions of specified state s
      {
        Get a transition t
          While not the end of transitions of the destination of state s
          {
            Get a tt transition
                    If (tt has an output action which strats with ( "!") && ( t has either input or delay action)
            {
                        Make a new NFSM_Transition nt from s to it's transition's destination with action
                        Input|Output
                    } End If

            If ( nt is not in the NFSM transitions of s)
            {
              Add it to the vector of s transitions

            } End If
          } End while not the end of transitions of the destination

            // add the NFSM-delay transitions
            Make a new NFSM-delay transition ntd from s to destination of tt with   action delay|-

          If ( ntd is not in the NFSM transitions of s)
            {
              Add it to the vector of transitions of s

            } End If

      } End while not the end of transitions of specified state s
   } End while not the end of Grid Automaton states

//add reset clocks as output actions
While not the end of NFSM states
   {
    While not the end of transitions of ss
      {
        While not the end of clocks of the t destination
        {
          If (the destination has a clock c whose value is zero)
          {
                    Add reset clock as output action
          } End If
        } End while not the end of clocks
      } End while not the end of transitions of ss
   } End while not the end of NFSM states
//add input actions for those state, which don't have to make the NFSM more //specified
While not the end of input list
  {
  While not the end of NFSM states
    {
    While not the end of transitions of s
      {
        If (t does have an input action)
          {
                    Break the loop of s transitions
          } End If

          If (all t's don't have i as input action) && (it is the end of s transitions)
          {
                    Make new transition with the destination of s and action of i
                    Add this transition to s transitions
          } End If
      } End while not the end of transitions of s
   } End while not the end of NFSM states
} End while not the end of input list
END
```

Figure 15: Transformation algorithm

We have to point that our tool doesn't consider the transformation of NFSM to observable NSFM, however, we have to introduce the concept behind it. NFSM is observable if for every state S in NFSM, and every input|output pair a|b, there is at most one transition from S with label a|b. ONFSM's are a typical class of NFSM's where a state and an input|output pair uniquely determine the next state, while a state and an input alone don't necessarily determine a unique next state and an output. Figure 16 is an example of not being ONFSM because state S0 with pair a|0

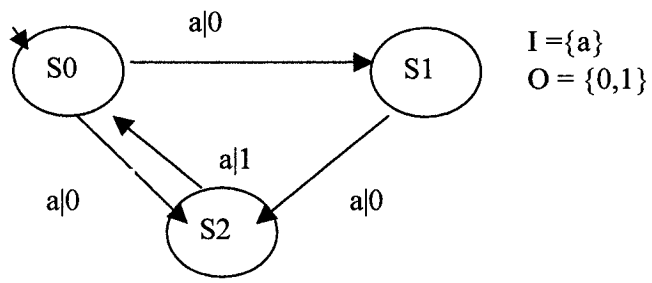

Figure 16: Not observable NFSM

determines two states which are not unique state, however Figure 17 is ONFSM of the previous figure since state S0 uniquely determines next state. In our testing method, it is assumed that the NFSM is observable.

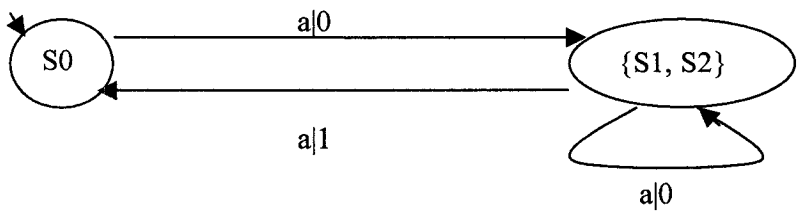

Figure 17: ONFSM of Figure 16

3.5 Minimizing the NFSM

The minimization of NFSM is the third step of our developed methodology. However, this minimization [34] is slightly different from the classical minimization of FSMs because it takes into consideration the time factor. In fact, instead of checking the equivalence for each pair of states, we consider only the pairs of states that have the same clock interpretations and different locations. The input for this Minimization process is NFSM information and the output is the minimized NFSM. How the tool minimizes the given NFSM is introduced in the following sections.

3.5.1 Minimization Process

Based on the NFSM information as an input, the equivalent traces are checked only for the states (l, v) and (l', v) such as l is not equal to l'. For that, we get all NFSM states, and for each pair of states (l, v) and (l', v), we check with a recursive manner if they accept the same traces. If one of the two accepts a trace, which doesn't belong to the other one, both of them are judged not equal. So, they are distinguishable by, at least, an entry sequence. However, if all traces of the two states are examined and none of them can distinguish a state from another, both of them are judged as equal states. In the case of the equivalence, we remove one of the two states and its children. Obviously, all the outgoing transitions from the removed state are deleted and all the ingoing transitions are oriented to the equivalent state [34]. The equivalent states are removed to avoid having redundant information. In Figure 18, (S1, 1) and (S2, 1) are equivalent states, since they have the same transition information (1|-). The same case for (S1, ∞) and (S2, ∞).

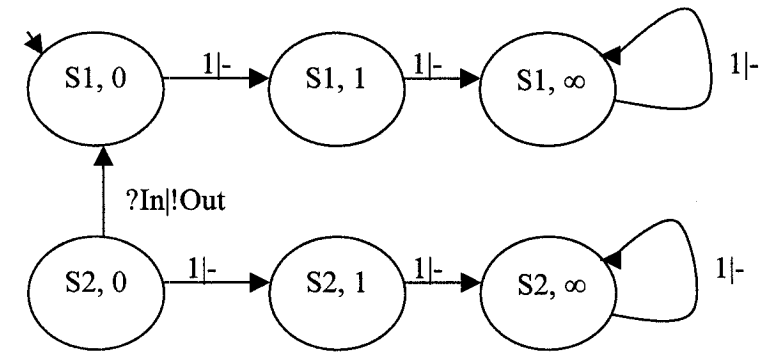

Figure 18: : Non-minimized NFSM

The Minimized NFSM of Figure 18 is represented in Figure 19. It is noticed that state(S1, 0) is not equivalent with the state (S2, 0) since (S2, 0) has a transition ?In|!Out, which (S1, 0) doesn't have.

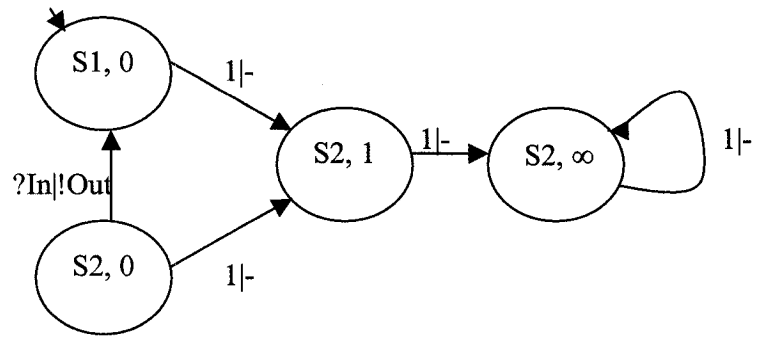

Figure 19: The minimized NFSM of Figure 18

3.5.2 Minimization Algorithm

In this section, we introduce the minimization algorithm supporting of what we have explained earlier. The time complexity of the algorithm is O(S*S*t*t*t) where S is the number of states and t is the number of transitions. We have used the linear search in

this algorithm, however, searching by sort is better to be used in such context because the search is done only for the states that have the same clock values but different locations and accordingly sorting the states is based on their clock values.

Input: a non-minimal NFSM
Output: a minimal NFSM

```
BEGIN
For (each state s1 of NFSM states)
{
      For (each state s2 of NFSM states)
            {
            If ((s1.location ! = s2.location) and (s1.clock = s2.clock))
            {
                  If (EqualStates(s1, s2) = true)
                        {
                        Attach input of s2 to s1 since they are equivalent
                        Delete s2 state and its children
                  } End if
            } End if
      } End for
} End for
END

// EqualStates(s1, s2) function
Function EqualStates(s1, s2) returns Boolean

If (there is no transition included in s1 and s2 )
{
      Return true
} End if

For (each transition of s1)
{
          EquiT = false
          EquiS = false
          Get t1 transition
      For (each transition of s2)
          {
            Get t2 transition
            If (t1.input = t2.input) && (t1.output = t2.output)
                  EquiT = True
                  If (s1=t1.destination) && (s2=t2.destination)
                  {
                        EquiS = true
                  } End if
                  Else
                        {
                        EquiS = EqualStates(t1.destination, t2.destination)
                  } End Else
                    } End if
      } End for
          If (EquiT = false) or (EquiS = false)
          {
            Return false
          } End if
} End for
Return true
```

Figure 20: Minimization algorithm

3.6 Generating the Test Cases

This is the fourth step of our developed methodology. Based on the minimized NFSM information as input, the generalized WP has the following five steps in order to produce the test cases as the output of the tool [6]:

1. Construct the state cover set (Q set). Refer to 3.6.1 section.

2. Construct the characterization W set and the state identification set Wi = {W0, W1... Wn-1} of S. Refer to 3.6.5 and 3.6.6 sections.

3. Construct the two sets P and R. Where P = Q.I, R= P/Q and I is the input list. Refer to 3.6.3 and 3.6.4 sections.

4. Generate the test suite one: $\Pi 1 = Q.\ I\ (m\text{-}n).\ W$. Refer to 3.6.7 section.

5. Generate the test suite two: $\Pi 2 = R.\ I\ (m\text{-}n) \oplus \{W0, W1... Wn\text{-}1\}$, where m >= n and m is the max number of states in IUT and n is the number of states in NFSM. That formula means that for each element in R set, find its corresponding reachable state from the initial state of NFSM by applying the same input sequences of R element. Then, calculate Wi for that reachable state. As a result of concatenating each element of R with its Wi, the test suite two is generated. Refer to 3.6.8 section.

6. $\Pi = \Pi 1 \cup \Pi 2$. That means the test suites is the union of the test suite one and the test suite two.

The first part checks that all the states defined by the specification are identifiable in the implementation. At the same time, the transitions leading from the initial state to these states are checked for the correct output and the state transfer. In this sense, this checking is a kind of the state verification.

The second part checks the implementation for all the transitions defined by the specification, which are not checked during the first part. It is a kind of the transition verification. Figure 21 illustrates NFSM for a system under test, which is completely specified [26].

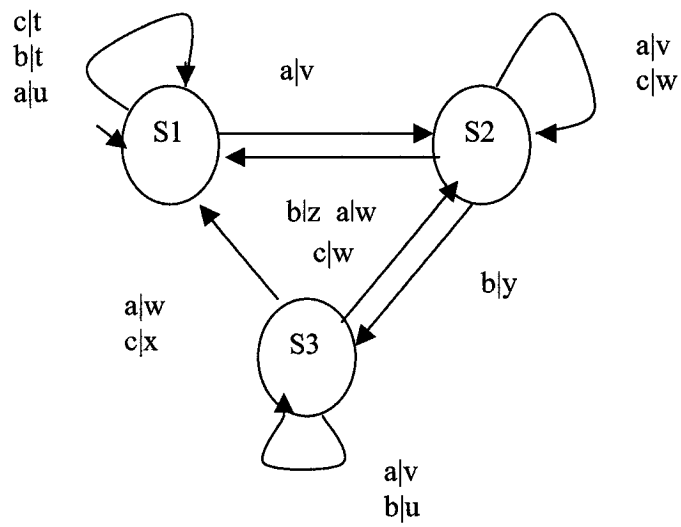

Figure 21: NFSM under test

Q = {ε, a, a.b} P = Q.I = {ε, a, b, c, a.a, a.b, a.c, a.b.a, a.b.b, a.b.c}
R = P/Q = { b, c, a.a, a.c, a.b.a, a.b.b, a.b.c} W = {b}
W1 = {b} W2 ={b}
W3 = {b}
∏1 = Q.w = {b, a.b, a.b.b}
∏2 = R⊕ {W0,....Wn-1} = b.W1 ∪ c.W1 ∪ a.a.(W1∪W2) ∪a.c.(W1∪W2)∪ a.b.a(W1∪ W2 ∪W3) ∪ a.b.b.(W1∪W3) ∪ a.b.c.(W1∪ W2) = {b.b, c.b, a.a..b, a.c.b, a.b.a.b, a.b.b.a, a.b.c.a}
∏ = {a.a.b, a.b.a.b, a.b.b.b, a.b.c.b, a.c.b, b.b, c.b}

3.6.1 Q Process

By recalling its definition, a Q set is a state cover set of NFSM, if for every state Si; Q contains an input sequence that brings the M machine from the initial state S0 to Si. Therefore, we need a tree that is drawn from the initial state (the root of the tree) to the other states of NFSM. In other words, finding the Q is like finding the shortest distance from the initial state to the rest of NFSM states. Dijkstra's algorithm is applied to find the Q set. It finds the shortest path from a chosen source to a given

destination. To apply the algorithm on NFSM states, then we have to think of NFSM states as a graph, which has the states as vertices (or nodes) and the transitions as edges that link the states together. The transitions are directed and have an associated *distance*, sometimes called the weight or the cost. We have the assumption that each transition has a weight of 1. The distance between the states *A* and the state *B* is noted [*A*, *B*] and is always positive 1 [35].

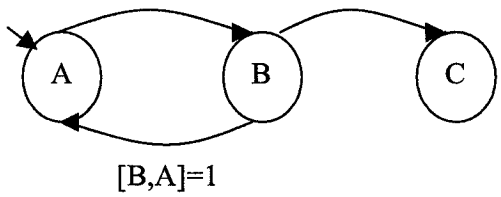

Figure 22: Simple graph

Dijkstra's algorithm divides the states into two distinct sets, the set of *unsettled* states and the set of *settled* states. Initially all the states of NFSM are unsettled, and the algorithm ends once all the states are in the settled set. A state is considered settled, and moved from the unsettled set to the settled set, once its shortest distance from the initial state has been found.

With these definitions, the algorithm is the following:

1. Initializing the *shortestDistances* to infinity.
2. Initializing the predecessors and *unsettledNodes* to empty sets.
3. Adding the initial state of the NFSM *s* to *unsettledNodes* set.
4. Mapping the shortest distance of *s* to value of "0".
5. While not the end of *unsettledNodes* set.
    - Getting the first element *f* of *unsettledNodes* set.
    - Adding *f* to *settledNodes* set.
    - Relaxing the neighbours of *f*.

How to relax its neighbours?

We go through each state "destination" adjacent to state *f* and see if the destination is not in the *settledNodes* and if the *shortestDistances* of "destination" is greater than the sum of the *shortestDistances* of *f* and [*f*, destination](which is 1), then:

1. Mapping the *shortestDistances* of "destination" with value of (*shortestDistances* of "f" + 1).
2. Making *f* as a predecessor of "destination".
3. Adding "destination" to *unsettledNodes* set.
4. then repeating number 5.

For example, we shall run Dikjstra's shortest path algorithm on the following simple NFSM states [35] (without specifying the transitions' information and the clock values of the states), starting at the initial state *a*. Figure 23 shows the simple NFSM graph.

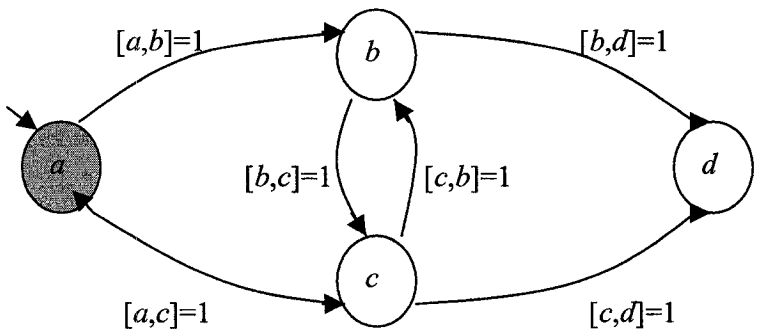

Figure 23:Dijkstra algorithm (1)

First, add the initial state *a* to *unsettledNodes*. Now *unsettledNodes* isn't empty, and the minimum is extracted, which is *a* again. *a* is added to *settledNodes*, then relaxing its neighbours. See Figure 24 to notice the relaxation of *a*.

Those neighbours are the states adjacent to *a*, which are *b* and *c*. The best distance estimate from *a* to *b* is computed. *shortestDistances*(*b*) was initialized to infinity, therefore:

- *shortestDistances(b)* = *shortestDistances(a)* + [*a*,*b*] = 0 + 1 = 1
- *predecessor*(*b*) is set to *a*, and *b* is added to *unsettledNodes*.
- Similarily for *c*, *shortestDistances*(*c*) is assigned to 1, and *predecessor*(*c*) is *a*.

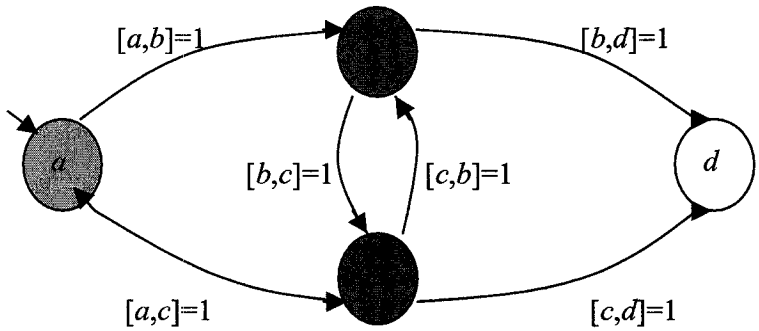

Figure 24: Dijkstra algorithm (2)

Now *unsettledNodes* contains *b* and *c*. *c or b* can be the state with the current shortest distance, which is 1. Let's say *c* is extracted from the queue and added to *settledNodes*, the set of settled nodes. Once more, relax the neighbours of *c*, which are *b*, *d* and *a*. See Figure 25.

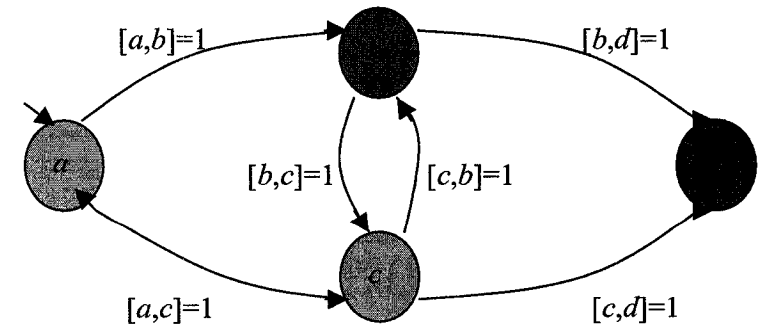

Figure 25: Dijkstra algorithm (3)

a is ignored because it is found in the settled set. It is clearly that the first pass of the algorithm had concluded that the shortest path from *a* to *b* was direct. Looking at *c*'s neighbor *b*, it is realized that:

shortestDistances(b) = 1 < *shortestDistances(c)* + [*c,b*] = 1 + 1 = 2

If *shortestDistances(b)*> *shortestDistances(c)* + [*c,b*] then the *shortestDistances(b)* will be updated by the value of (*shortestDistances(c)* + [*c,b*]) and the predecessor is updated to *c*.

The next adjacent state is *d*. *shortestDistances*(*d*) is set to 2 and *predecessor*(*d*) to *c*.

The unsettled state with the shortest distance is extracted from the queue, it is now *b*.

It is added to the settled set and its neighbor *c* is relaxed.

c is skipped, it has already been settled.

At this point the only state left in the unsettled set is *d*, and all its neighbors are settled. See Figure 26.

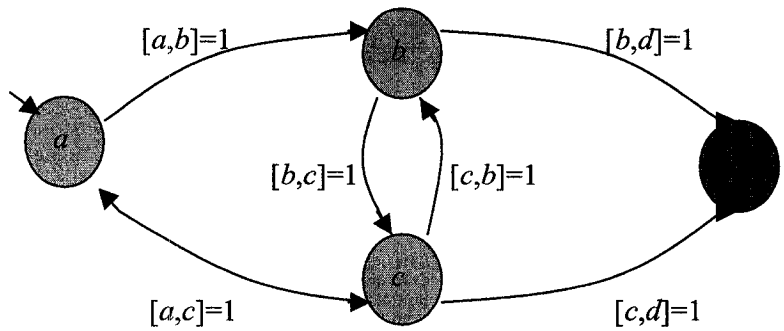

Figure 26: Dijkstra algorithm (4)

The algorithm ends. The final results are displayed in the straight arrows below:

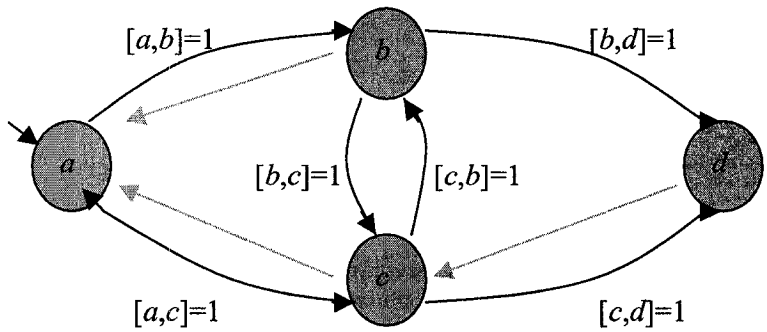

Figure 27: Dijkstra algorithm (5)

3.6.2 Q Data Structure

According to the above explanation, the data structures that are needed for the Q algorithm (Dijkstra Algorithm) are in the following table [35]:

Table 5: Dijkstra data structure

Class	NFSM_States
Attribute	*settledNodes*
Data Structure	Set
Rationale	*settledNodes* contains the set of the settled states in NFSM, whose shortest distances from the source have been found. The Java collection includes the Set interface, and more precisely, the *HashSet* implementation. It offers the constant-time performance on the "contains" operation, which is the only needed one. Set is an abstract type is declared as the data structure of *settledNodes* instead of concrete type (*HashSet*). Doing so is a good software engineering practice, as it allows changing the actual type of the collection without any modification to the code that uses it.
Attribute	*shortestDistances*
Data Structure	Map
Rationale	*shortestDistances* stores the best estimattion of the shortest distance from the source (the initial state of the NFSM) to each state in NFSM. "Map" is used to keep the shortest distance value for each state in the NFSM. As a result, it is a mapping from each Location object to an integer value. Map is an abstract type.
Attribute	*predecessors*
Data Structure	Map
Rationale	*predecessors* stores the predecessor of each state in the NFSM on the shortest path from the initial state. "Map" is used to store the one-to-one relationship between two states, one state is a predecessor of another state in order to keep the track of the shortest path. As a result, it is a mapping from each Location object to another Location object.
Attribute	*unsettledNodes*
Data Structure	SortedSet
Rationale	*unsettledNodes* contains the set of the unsettled NFSM states. This structure keeps the set ordered all the time in order to get the first element with the lowest distance. However, a comparator is needed to order the states in the *unsettledNodes*.
Attribute	*ShortestDistanceComparator*
Data Structure	Comparator
Rationale	This attribute is used to insert any new element in the right place in the *unsettledNodes* set.

3.6.2 Q Algorithm

After we have explained the data structure that is required for the Q algorithm, the Q algorithm is introduced in this section. The time complexity is O(S+S*log(S)) where S is the number of states.

Input: NFSM_States object
Output: Q set which is a vector of vectors
Initially:

- shortestDistances is infinity
- predecessor is empty
- settledNodes and unsettledNodes are empty sets
- Q is an empty vector

```
BEGIN
//find the shortest distance for each state using the Dijkstra algorithm
Add the initial state s of NFSM to unsettledNodes

While (unsettledNodes is not empty)
{
state_one = extract_minimum(unsettledNodes)
            Add state_one to settledNodes
            relax_neighbours(state_one)
} End While

// Get the Q set by extracting the input action from each transition of the shortest distance
// that is determined by the above predecessor for each state
While (not the end of NFSM)
{
Get a NFSM state s
While (not the end of predecessor of s)
{
Get a predecessor
Get the input from each predecessor to its corresponding state
Store the input in a vector input
} End While
Reverse the input vector
Store the reversed input vector in a Q vector
} End While
Return Q
END

//relax_neighbours(state_one) function
For each state s adjacent to state_one
{
If (state_one is not in the settledNodes)
{
If ( shortestDistances(s) > (shortestDistances(state_one) + 1))
                        {
                        shortestDistances(s) = (shortestDistances(state_one) + 1
                        Predecessor = state_one
                        Add s to unsettledNodes
                        } End If
} End If
} End For

//extract_minimum(unsettledNodes) function
Find the smallest (as defined by the shortestDistances) state in unsettledNodes set
Remove it from the set
Return it
```

Figure 28: Q algorithm

3.6.3 P Algorithm

By recalling the definition of P set: P set is a transition cover if, for every transition si → sj with a|b action, P contains input sequences α and α.a that bring the M machine from the initial state s0 to si and sj, respectively. It can be simply obtained by concatenating the Q set with the Input list. This will cover the transitions, where P set is necessary for obtaining R set as it was explained in 3.6 section. This needs a time complexity O(I*Q) where I is the number of input elements and Q is the number of Q elements.

Input: Q vector and Input vector
Output: P vector
Initially: Vector P is empty

```
BEGIN
While (not the end of Q set)
{
Get a vector "in1" from Q vector
"in11" = clone "in1"
While (not the end of Input set)
{
Get a vector "in2" from Input set
"in22" = clone "in2"
Add "in22" vector at the end of "in11" vector
} End while not the end of Input vector
Add "in22" to P vector
} End while not the end of Q set
Return P vector
END
```

Figure 29: P algorithm

3.6.4 R Algorithm

R set denotes all transitions of NFSM except those belonging to Q set. Therefore, the Q transitions are removed from P set to obtain R set. We need R set to calculate the test suite two as it is explained in 3.7 section. R algorithm needs a time complexity: O(Q) where Q is the number of Q elements.

Input: Q vector and P vector
Output: R vector
Initially: R vector is empty

```
BEGIN
R = clone of P vector
While (not the end of Q set)
  {
    Get a vector "in" from Q vector
          If (R contains "in" vector)
        {
          Remove "in" vector from the R vector
         } End If
  } End while not the end of Q set
Return R
END
```

Figure 30: R algorithm

3.6.5 Wi Algorithm

State Identification set (Wi) is defined as a given Si state, for any other state Sj, there must exist an input sequence in Wi such that two different sets of possible output sequences are produced when this input sequence is applied to their states. Wi set is important to produce both test suites. When we calculate Wi for each state, we start with the input list I. If it is not sufficient then we grow up with I.I. If it doesn't produce different outputs then we try I.I.I and so on. Input list of NFSM and NFSM information are needed to calculate the Wi. In Figure 31, the Wi algorithm is introduced and its time complexity is O((I*S)+(I*I)) where S is the number of the states and I is the number of the input elements. Again as in the minimization algorithm, we have used linear search here. It is better to use the sort search since the search is done through the states, which have the same clock values and different locations.

Input: Input vector, NFSM state s and NFSM_States object
Output: Wi vector of strings
Initially: Wi is empty vector

```
BEGIN
//notice that two states must have different locations and same time as a condition //inside check function
While (not the end of Input elements)
 {
   Get an input element "in"                    //input element is a vector of strings
   If (check(s, NFSMStates object, in)) // check if they have different output for the   same input
   {
     Wi = in
     Return Wi
   } End If
 } End while not the end of Input elements

//If there is no input element in Input vector that gives different output then call
//Wi recursively with new Input vector, which is equal to Input.Input

Create vector list
list = Input.Input
Wi = get_Wi(list,s,NFSM_States object)
END
```

Figure 31: Wi algorithm

3.6.6 W Algorithm

W is the union of all Wi. It is needed to produce the test suite one as it was shown in 3.6 section. The test suite one is W.Q. In Figure 32, the W algorithm is introduced. And its time complexity is O(S) multiplied with the Wi time complexity where S is the number of states.

Input: NFSM_States object and Input vector
Output: W vector
Initially: W vector is empty

```
BEGIN
While (not the end of NFSM states)
 {
   Get a NFSM state s
   Create Wi vector
   Wi = get_Wi of s
   If (W doesn't contain Wi of s)
    {
      W.addElement(Wi)
    } End If
 } End While not the end of NFSM states
Return W
END
```

Figure 32: W algorithm

3.6.7 Test Suite One Algorithm (Q.W)

As it was explained in 3.1 section, test suite one and test suite two are the desired results of our tool. Calculating the test suite one depends on Q set and W set information. Its algorithm is simple as it is shown in Figure 33. The time complexity of the algorithm is O((Q*W)+(T*T*E)) where Q is the number of Q elements, W is the number of W elements, T is the number of test suite one elements and E is the number of one test element sequences. For example the E of (In. ½ .In) is equal to 3.

Input: Q vector and W vector
Output: Test Suite One vector
Initially: test vector is empty

```
BEGIN

// It is important to say that the test suite one must be optimized to avoid redundant test //cases

test = Concatenate(Q,W)
test = optimize(test)
Return test
END
```

Figure 33: Test suite one algorithm

3.6.8 Test Suite Two Algorithm (R.Wi)

Another important output of our tool is the test suite two. It can be calculated depending on R set and Wi set information. Its algorithm is shown in Figure 34. The time complexity of the algorithm is O(R*R*t*t*s*s*Wi) multiplied with the Wi time complexity where R is the number of R elements, t is the number of the transitions, s is the number of the states and Wi is the number of Wi elements.

Input: R vector, NFSM_States object and Input vector
Output: test suite two vector
Initially: test vector is empty

```
BEGIN

While (not the end of R)
  {
   Get r element from R
   Get the initial state s from the NFSM_States object
   While (not the end of transitions of s)
   {
      Get a transition t
      If (t.input.equals(r.elementAt(0)))
      {
           //call function check, which calls itself recursively to obtain the vector of reachable states from the initial
          //state of NFSM by applying the input sequence of element r
            vector of NFSM states = check(r)
            While (not the end of states)
             {
             Get state ss from states
             Get Wi for ss
                    Store it in Wi vector
              } End while not the end of states
      } End If
   } End while not the end of transitions of s
   Concatenate r with each element of Wi vector using function of Concatenate
   Add the concatenation to the vector test
  } End while not the end of R
Return test
END
```

Figure 34: Test suite two algorithm

3.7. Optimization the Test Cases

Finally, the tool optimizes the number of the test cases if they are large. One approach is to delete any test case, which is a prefix of another one. Another way can be generating test cases only for the critical parts of the real time systems. The last approach to think of is using large granularity or selecting one representative of a clock region to optimize the large number of the generated test cases.

3.7.1 Optimization Algorithm

Using the first approach, the optimization algorithm is as in Figure 35. The input of this algorithm is the test suite one or the test suite two. The time complexity of the

optimization algorithm is O(T*T*E) where T is the number of the test suite elements and E is the number of the test element sequences.

Input: not optimized test vector
Output: optimized test vector

```
BEGIN
//Optimization of the test cases by removing the prefix
While (not the end of test vector)
  {
   Get an element from test vector: element1
   While (not the end of test vector clone)
     {
      Get an element from test vector clone: element2
       If (the position of element1 != the position of element2)
       {
          If (element1.size () < element2.size ())
           {
              While (not the end of element1)
               {
                              If (one element of element1 != one element of element2 of same position)
                  {
                     Break of element1 loop
                  } End If
                  Else if(the end of element1 loop)
                          {
                     Remove element1
                  }
               } End while not the end of element1
           } End If
       } End If
     } End while not the end of vector
  } End while not the end of vector
Return test vector
END
```

Figure 35: Optimization algorithm

3.8 Conclusion

In Chapter 3, the developed methodology was detailed. The TIOA specification is the input and the timed test cases are the output of the tool. To generate the test cases, the tool parses the TIOA specification in order to construct its corresponding Grid Automaton that will be transformed by the tool into NFSM. The tool also minimizes the NFSM and accordingly it generates the test cases. The optimization of the test cases is also done by the tool in order to avoid the large number of the test cases. Each step is explained with examples and their algorithms are introduced. Moreover, we

have used [36] and [37] to calculate the time complexity of the proposed algorithms. They are considered big since the algorithms are exponential on the number of the clocks, and the constants used as bounds in the time constraint. Appendix B shows the activity diagrams for more details about the algorithms, using a concrete context.

In next chapter, we put the developed methodology into the implementation framework. The design issues such as the high level language, the user interface, the parser, the tool parts...etc will be also covered in next chapter.

CHAPTER IV: TOOL IMPLEMETATION

In this chapter, the major design decisions and their solutions, the class diagram and the sequence diagram are explored so that the included information is clear, consistent and correct with the previous chapters. It is important to mention how the class interface look like, how the relationships introduced in the class diagram implemented, and how the tester interacts with the user interface of the tool. The scenario of execution for the tool is introduced as well.

4.1 Design Rationale.

This section summarizes many of the issues, which need to be addressed or resolved in order to devise a complete design solution. It also addresses the main issues of the detailed design of the tool

4.1.1 User Interface

This tool, like most of today's applications, has a graphical user interface. This facilitates the interaction of the users with the application, but at the same time, adds some complexity and difficulty to the design of the tool. In the following table, we have introduced some of the available alternatives and we have chosen one of them according to some reasons that are also stated in the table.

Table 6: User interface issue

Issue 1	User interface of the tool.
Alternative 1	Interweave the user interface with the core functionalities of the application.
	Benefits: - Simplifies the complexity of the design. Liabilities: - The reusability of the various components is limited.
Alternative 2	Use the architecture, which build the programs that have a graphical user interface: the model-view-controller design pattern (MVC). As its name indicates, the MVC paradigm breaks an application into three components. The model contains only the data and the functionalities that are related by a common purpose. The view is responsible to display the information to the user. The controller accepts input from the user and instructs the model to perform actions based on that input. The controller and the view, combined together, form the user interface. Furthermore, a change-propagation mechanism ensures consistency between the user interface and the model [38], [39], [40].
	Benefits: - Facilitates the reuse of the components. Liabilities: - Following the MVC structure tends to increase the complexity of the design and its corresponding implementation. - The controller and view are separate but closely related components, which hinders their individual reuse.
Selected Solution	The chosen alternative for this tool is the first one, since its advantages outweigh its liabilities. Although the MVC is a very common and powerful architecture, the tool is only seeking a simple interface that allows the users to interact with it in friendly way rather than running the tool in Unix environment. The tool interface is not for reusability issues but it is more about seeing the results in the applet.

Therefore, the tool has a very simple graphical user interface as shown in Figure 36. It is just for seeing the results of the tool by clicking the desired button. By clicking the "TIOA" button, the information of TIOA specification that the tester has just entered

will be displayed. The "Grid Automaton" and "NFSM" buttons are for displaying the Grid Automaton and NFSM information. If the tester is interested to see the Q, P, R and W sets, then s/he clicks the "Q set", "P set", "R set" and "W set" buttons. Finally, the most important buttons to display the test cases are "Test suite 1" and "Test suite 2" buttons.

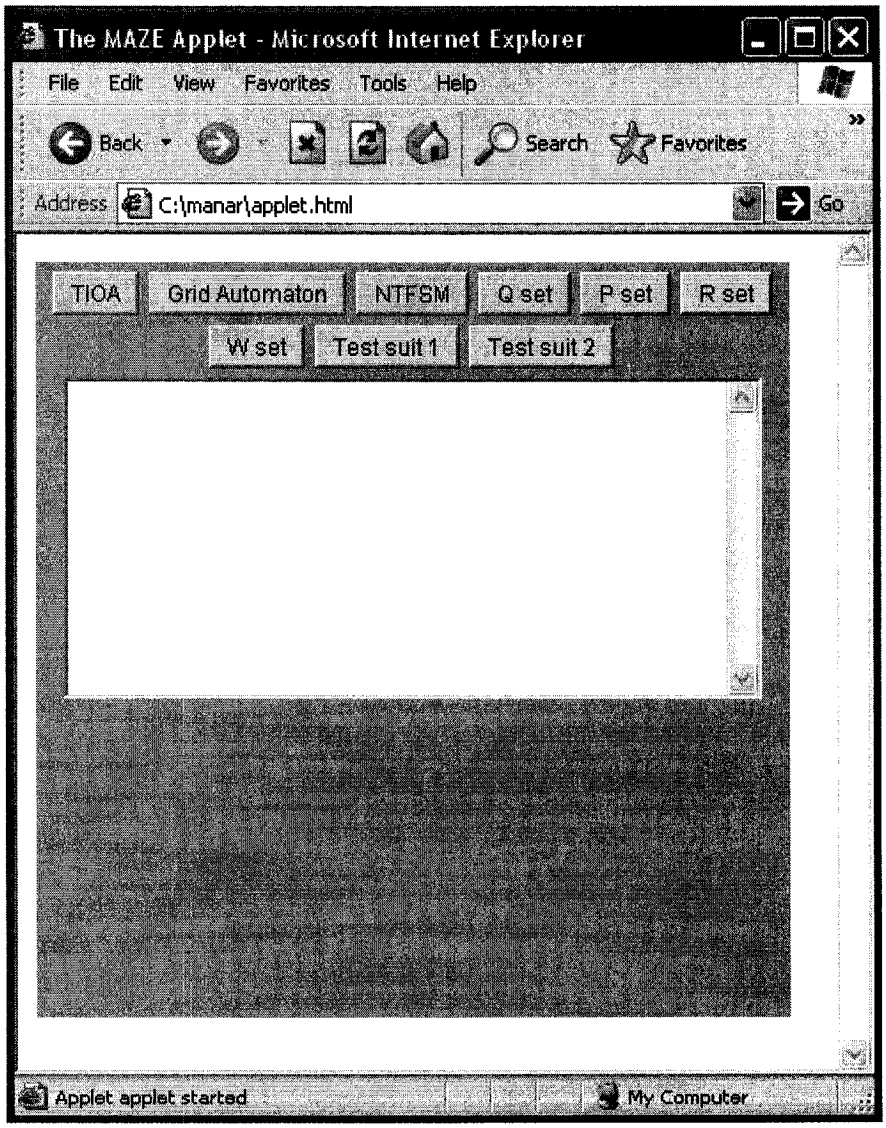

Figure 36: Tool user interface

4.1.2 Why JAVACC?

As mentioned before, the tool needs a parser to parse TIOA file, which is written by the tester to describe the real time system. In table 7, two alternatives are shown and we have chosen Java Compiler Compiler (JavaCC) as the parser of our tool.

Table 7: Parser issue

Issue 2	The parser of the tool.
Alternative 1	Interweave the tool with a function that reads the file by using while and for loops.
	Liabilities: - The function must be so precise to describe the format of the file. For example, the spaces between the words and the new lines must be taken into consideration. - Many loops are needed to read the file. - Memory leak problem will happen if the file size is big.
Alternative 2	Use a compiler that has a simple and readable structure [41], [42, 43].
	Benefits: - A simple compiler will neither need any code improvements (optimization) nor create the parse tree explicitly - No need for the loops. It is the responsibility of the lexical analyzer to scan the file and identify the groups of the characters that form the tokens.
Selected Solution	Using a compiler is better since it can generate the code for individual components of a file at the moment at which the corresponding syntactical structures are recognized.

For a program to receive an input, either interactively or in a batch environment, another program or a routine is provided to receive the input. Complicated input requires additional code to break the input into pieces that mean something to the program [41], [42]. **Java Compiler Compiler (JavaCC)** is used to develop this type of input program. JavaCC generates a lexical analyzer program that analyzes the input, which is a sequence of character, and it is represented by a Java *InputStream* object or a Java *Reader* object. It is the first phase in a compiler, which reads the input

source and converts the character strings in the source into individual *Token* objects. The tokens are defined by the grammar rules set up in the *.jj* specification file. Using the regular expressions and BNF productions, we can specify patterns to *.jj* that allow scanning and matching strings in the input. Each pattern in *.jj* has an associated action, which is a fragment of C code that performs some action. Typically an action returns a token, representing the matched string, for subsequent use by the parser [44], [45].

JavaCC also generates a parser program that analyzes the input using the tokens identified by the lexical analyzer. As the lexical analyzer reads the source file, the initial characters in the remainder of the input stream are checked against the legal token strings. These tokens strings are checked in the order in which they are listed in the lexical analyzer. The analyzer attempts to match the longest segment possible from the input. The output of the parser is whatever the programmer wants it to be, as long as it can be expressed in Java. In our case, the output is reading the file and storing its information to its corresponding data structure of TIOA object [44], [45].

To that end, the compiler has the following structure [42]:

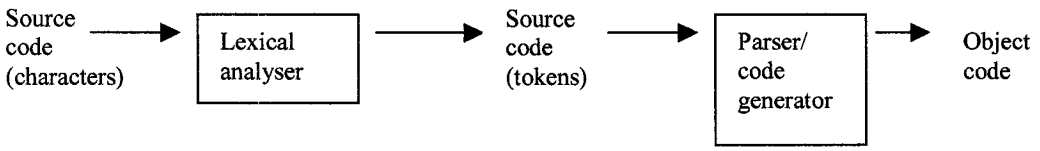

Figure 37: Compiler structure

The tokens of the file, which have the same format described in the previous chapter, must be specified in the following manner:

Space must be specified so that it is skipped and it is not passed to the parser, while reading the file. Line breaks have different representations depending on the operating system. For example, in Unix and Linux, a new line character "\n" is used, in DOS and Windows a carriage return "\r" followed by a new line is used, and in other operating systems a carriage return alone is used. They are defined as tokens. Other tokens must be specified to represent the information of the file such as S, which represents the source of a state (or location), D, which is the destination of a transition of a specific state, etc. The tokens are important to be passed on to the parser and in turn the parser determines the structure of the program [45]. The following are the tokens of TIOA file:

```
SKIP : { " " }
TOKEN : {
      < EOL : "\n" | "\r" | "\r\n" >
  |   < ENDINPUT : "#####" >
  |   < ENDSTATE : "###" >
  |      < ENDTRANSITION : "#" >
  |   < TRAN : "transition" >
  |   < CONSTRAINT : "constraint" >
  |   < RESET : "reset" >
  |   < Source : "S" >
  |   < Desti : "D" >
  |   < special : "?" | "!" | "=" | ">" | "<" >
  |      < Others : (<LETTER> | <DIGIT> | <special> )+  >
  |      < LETTER : (["a"-"z", "A"-"Z"])+  >
  |      < DIGIT  : (["0"-"9"])+  >
}
```

Now the regular expressions are defined. For example, the generated method "TIOA_State State()" in the next page is BNF production of TIOA state. The return type of that method is TIOA_State. The variables are then declared inside the method State() such as the vector of the transitions, the vector of the time constraints, the vector of the clocks and the variable "t", which is of type Token. Token is a generated class that represents tokens.

The BNF production section for State() is followed after that. It is noticed that within the braces in a BNF production, Java statements are added. Now, when a token is matched in BNF production, the Token object is recorded by assigning a reference and the image field of the Token class records the matched string of the characters. The meaning of the star "*" in BNF production of TIOA_State is having one or more transitions. The other methods are having the same structure as below one. (See Appendix A for more details about the other methods.

```
TIOA_State State() :
{
        Token t;
        String source;
        TIOA_State state;
        TIOA_Transition transition;
        Vector trans;
        Vector consts;
        Vector resClk;
}

{
    {trans =  new Vector();}
    {consts =  new Vector();}
    {resClk =  new Vector();}

    < TRAN >
    < Source >
        t = <Others>
    {source = t.image;}
    transition = Transition()
    consts = constraint()
    resClk =  reset()
    { transition.setCLK( resClk);  }
    { transition.setConst( consts); }
    {trans.addElement(transition);}
    (
        <ENDTRANSITION>
        <EOL>
        transition = Transition()
        consts = constraint()
        resClk =  reset()

        { transition.setCLK( resClk);  }
        { transition.setConst( consts); }
        {trans.addElement(transition);}
    )*
    <ENDSTATE>
    <EOL>
    {state = new TIOA_State(source, trans);}
    { return state;}
}
```

4.1.3 Why JAVA?

The tool is more efficient to be implemented in an Object Oriented Language. C++ and Java are two alternatives. In the following table, we discuss their benefits and liabilities.

Table 8: High level language issue

Issue 3	Choosing the high level language.
Alternative 1	C++.
	Benefits [46]: - It provides the generic classes' facility (templates), which refers to the ability to parameterize a class with specific data types. - It gives the ability for a programmer to overload the operator such as +, * ...etc. - C++ programs generally run faster than Java programs. Liabilities [46]: - The programmer is responsible of freeing any memory that is no longer needed.
Alternative 2	Java.
	Benefits [46]: - It is strongly object oriented since all user-defined types are Objects. - There is a garbage collection mechanism, which frees the memory of unused objects. - There is no pointers arithmetic. - Java attempts to be simple and familiar without the loss of functionality or performance [43]. - Portability: programmers can be assured that their programs will run the same on many different platforms, as well as allowing the programmer to fix many of his/her errors early on as opposed to after distribution [43]. - Performance: Java is fairly fast for most applications in interpreted mode [43]. Liabilities [46]: - It is not having the ability to overload the operators and it can't provide templates of other data types different from the specified one.
Selected Solution	The tool is better to be implemented in Java since the memory leak problem will be avoided especially many large graphs will be used.

4.1.4 Loops Issue

How can we deal with the loops in the minimization algorithm? To avoid the deadlock loop while minimizing NFSM, the vector data structure is used to store the history of the checked transitions. When the transition that is contained in a loop in NFSM is checked and stored in the history vector, then it won't be checked again.

4.1.5 Determinism Issue

The tester must check that TIOA specification is deterministic before writing the file. Otherwise, an error message is displayed in the screen. The tool can't handle the non-deterministic TIOAs.

4.1.6 Q Algorithm

As it was mentioned earlier in Chapter 3, we have chosen Dijkstra algorithm to find Q set although there are other shortest path algorithms. In the following table, we compare between Bellman Ford algorithm and Dijkstra algorithm.

Table 9: Q issue

Issue 6	Choosing an algorithm to find Q.
Alternative 1	Bellman Ford Algorithm [47].
	Benefits: - It allows negative edge weights. - The time complexity is O(VE) where V is the number of vertices and E is the number of edges.
Alternative 2	Dijkstra Algorithm [47].
	Benefits: - All edge-weights are nonnegative. - The time complexity is O(E + V log V), where V is the number of vertices and E is the number of edges.
Selected Solution	The tool is better to implement the Dijkstra algorithm since it beats Bellman-Ford when there are no negative edge weights. Assuming that each edge (transition) in the NFSM has a weight of 1. In addition, the time complexity of Dijkstra algorithm is better.

4.2 The methodology used in designing the tool

4.2.1 Object Oriented Design Methodology

To design the real time system tool, the Object-Oriented Design methodology [48] is chosen to describe the solution to the testing problem in terms of objects. An object-oriented design consists of a set of *classes* and *data* associated with these classes, and the set of *actions* the class can be asked to undertake.

The object-oriented design methodology consists of the following steps:

1. Identify the classes.

   - Identify a large set of candidate classes such as TIOA, Grid Automaton, NFSM, State, Transition, Clock, and Action ...etc. Refer to the class diagram in this section.

   - Find a subset that provides a meaningful solution to the problem at an appropriate level of abstraction. For example, TIOA States, Grid Automaton States, NFSM States and the parser, where these classes consist of other classes.

2. Characterizing the class.

   - Provide a verbal description of each object. For example, TIOA States object has states and each state has transitions.

   - Define the data associated with the objects. For instance, the TIOA transition may has clock reset, time constraints and an action.

   - Avoid too many details.

3. Defining the implementation.

   - Finds an action that is certain to be part of the final implementation, e.g. main ().
   - Outline how each selected action will be implemented, e.g. pseudo code. Refer to Chapter 3.
   - Adds actions to related classes when exploring.

4. Pseudo code.

4.2.2 Overview of Tool's Class Diagram

In general, the tool for the real time systems consist of four packages based on a state characterization technique: TIOA_States, Grid_States, NFSM_States and the Parser. Each package has its own responsibilities and relatively independent. They are limited in contacting each other. The whole tool is flexible and extensible thereby. That's why we take advantage of vector framework to construct our tool. The tool is not a complex tool since its modules are simple to understand and its behavior is predictable according to its well-defined rules (See the class diagram). That is beside it is well documented. Therefore, these attributes ease the maintenance of the tool. The tool is also portable since java language is itself portable across operating systems and different architectures. Java uses universally standard and static data types [43].

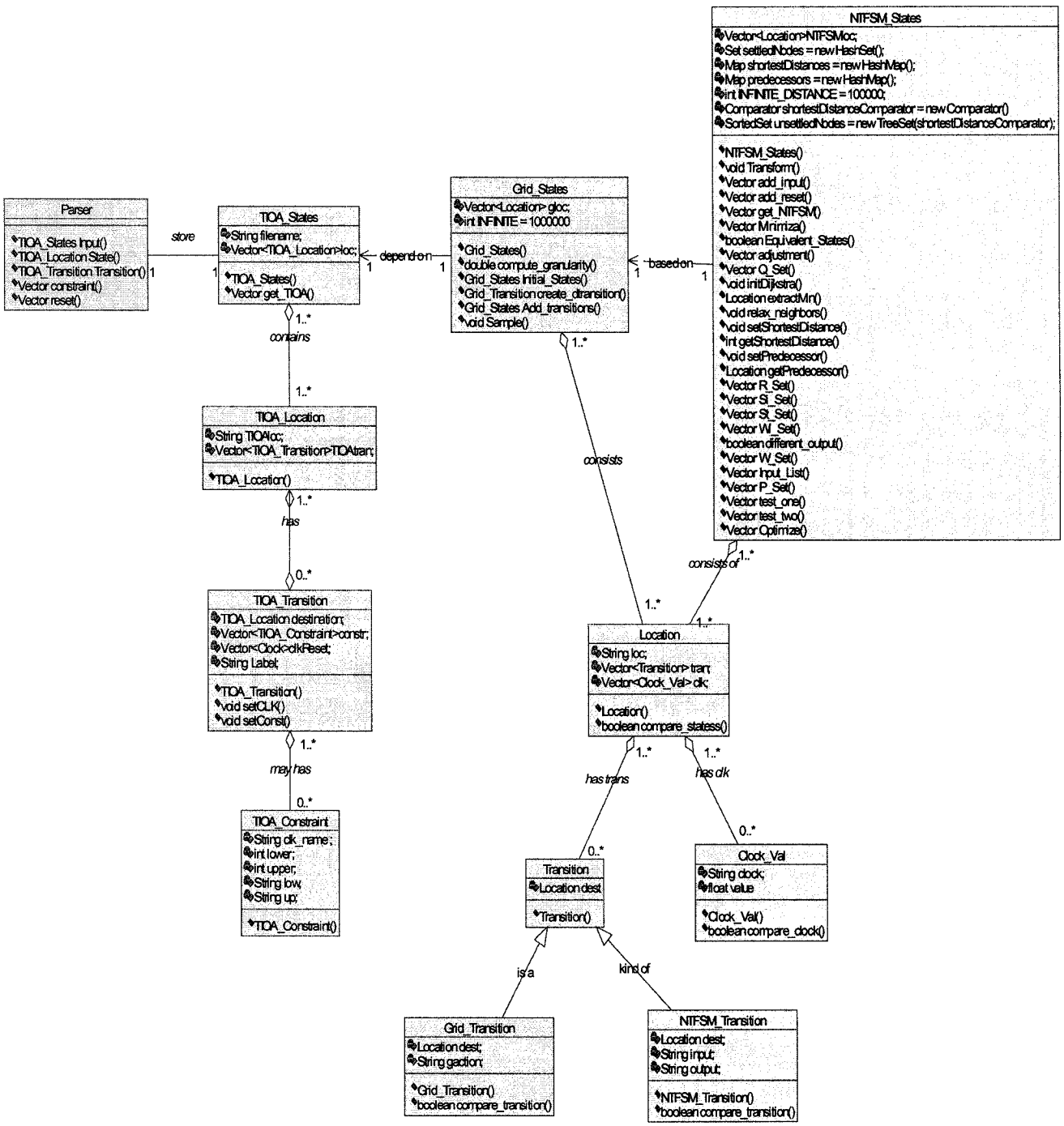

Figure 38: The class diagram of the tool

4.2.2.1 TIOA Package

TIOA package is to read the file entered by the user to construct the corresponding TIOA states in terms of vectors. It allocates the required memory for such automata. Detailed information of the classes in this part is shown in the table below.

Table 10: TIOA package

Class Name	Type	Description
TIOA_States	Super Class	is the main class that the second and third parts depend on it. It is responsible for reading the file. It is a vector list that contains the states of TIOA automata.
TIOA_State	Class	is a part of TIOA_States that represents the state of TIOA .It is characterised by the state name and its vector list of the transitions.
TIOA_Transition	Class	is a part of TIOA_State that represents the information of a transition for a specified state.
TIOA_Constraint	Class	is a part of TIOA_Transition that represents the time constraint for certain transition.

4.2.2.2 Grid Automaton Package

Grid Automaton package is like a bridge between TIOA package and NFSM package. It samples the given TIOA automaton using a suitable granularity, in order to construct a sub automaton that is easily testable, called Grid Automaton. It also allocates the required memory for that Grid Automaton. Detailed information of the classes in this part is shown in table 11.

Table 11: Grid Automaton package

Class Name	Type	Description
Grid_States	Class	is an important class which NFSM_States depends on. It is responsible for sampling the TIOA to Grid Automata. It is a vector list that contains the states of Grid Automata.
State	Class	is a part of Grid_States that represents the state of Grid Automata .It is characterized by the state name ,its time and its vector list of the transitions that comes out from it.
Grid_Transition	Class	is a part of State that represents the information of a transition for a specified state in Grid Automata.
Clk	Class	is a part of State that represents the information of clock for specified state.

4.2.2.3 NFSM Package

NFSM package is responsible for transforming the given Grid automata to a (NFSM). It allocates the required memory for that NFSM. It also minimizes the given NFSM in order to ease the operation of generating the test cases. Moreover, It handles all operations associated with the Generalized Wp-method to generate timed test cases from the minimized NFSM. Detailed information of the classes in this part is shown in table 12.

Table 12: NFSM package

Class Name	Type	Description
NFSM_States	Class	is the required class to obtain NFSM for test cases generation. It is a vector list that contains the states of NFSM.
State	Class	is also a part of NFSM_States that represents the state of NFSM .It is characterised by the state name ,its time and its vector list of the transitions that comes out from it.
NFSM_Transition	Class	is another part of State that represents the information of a transition for a specified state in NFSM.
Clk	Class	is a part of State that represents the information of clock for specified state.

4.2.2.4 Parser Package

Parser package is responsible for parsing the file that is send by TIOA_States.

Table 13: Parser package

Class Name	Type	Description
Parser	Class	Is the required class to parse the TIOA file and store its information in TIOA_States class.

4.3 Sequence Diagram for Tool Execution

The following sequence diagram shows the course of the tool execution. That is supporting to what we have said in Chapter 3. Table 14 explains the sequence diagram.

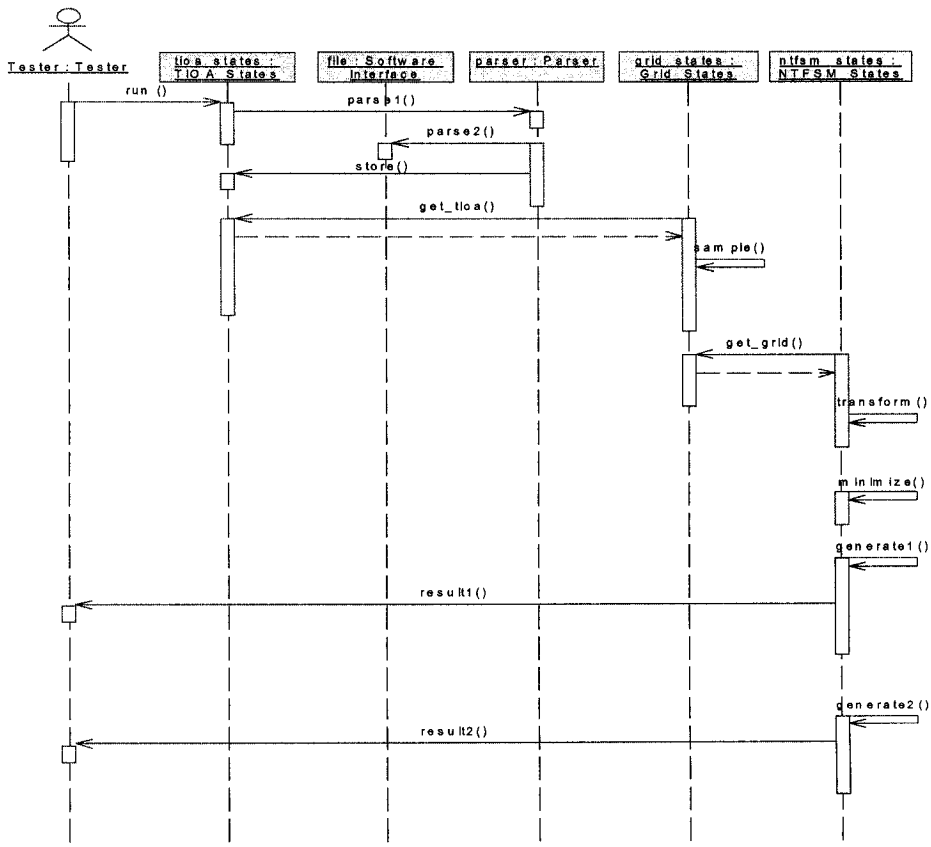

Figure 39: The sequence diagram of the tool execution

Table 14: Sequence diagram information of the tool execution

Interaction Diagram Name	Running the tool
Responsibility	This diagram is used to show the interaction of scenario for Use Case: Run the tool (Use Cases: Parse the TIOA file, Sample into the Grid Automaton, Transform into the NFSM, Minimize the NFSM and Generate the test cases are included).
Objects	**Origin**
:tioa_states	See *Class Diagram TIOA_States*
:grid_states	See *Class Diagram Grid_States*
:NFSM_states	See *Class Diagram NFSM_States*
:parser	See *Class Diagram Parser*
:file	See *Sequence Diagram Software Interface*
Sequential Actions	**Description**
1. run()	The user runs the tool.
2. parse1()	The tioa_states asks the parser to parse the TIOA file.
3. parse2()	The parser parses the TIOA file.
4. store()	The parser then stores the file information into tioa_states.
5. get_tioa()	grid_states asks tioa_states for TIOA information.
6. sample()	grid_states builds the corresponding Grid Automaton based on returned information.
7. get_grid()	NFSM_states asks the grid_states for the Grid Automaton information.
8. transform()	NFSM_states transforms the Grid Automaton to its corresponding NFSM based on the returned Grid Automaton information.
9. minimize()	NFSM_states minimizes NFSM.
10. generate1()	NFSM_states generates the test suite 1.
11. result1()	NFSM_states returns the test suite 1 to the user to analyze it.
12. generate2()	NFSM_states generates the test suite 2.
13.result2()	NFSM_states returns the test suite 2 to the user to analyze it.

4.4 Design Quality and Design Evaluation

4.4.1 The Five Factors That Effect the Design Quality

To have a good quality of design, there are five factors that must be considered [48]:

1. The relative level of detail.

- **The level of detail or granularity of the various components should be about the same.**

It is noticed that in the previous class diagram, the detail level of the variety classes is almost the same. The TIOA_States, Grid_States, NFSM_States and the Parser are the classes of the first level. They are describing the four parts that our tool is consisting of and the first three of them have its own responsibility toward the states it may have. Then, the second level of detail is the states of each part which are characterised by certain features that make it unique from other states. Finally, the third level is describing the details of each state's transitions, its information that may have on and their clocks. The last level is necessary for the first part, which is the constraint class that describes the details of constraint in which TIOA transition has.

2. The number of components at each level

- **Ideal number ranges from 5 to 10.**

Since our tool is quiet medium and it is not considered as a huge tool, then having three to four classes in each level is quiet fair to not have the complex relationships among classes. Moreover, it is enough to build co-operative components to construct the tool.

3. The final level of detail

- **The hierarchy should be terminated at an appropriate level depending on the audience of the design. I.e. reader feels comfortable to implement the design.**

The tool is broken into four parts to represent the main four steps in the test cases generation. Then, each part is broken into smaller pieces to handle the operations in easier way and to not find the hardships during the implementation of the tool. These smaller pieces can be directly mapped to a code in any implemented language.

4. Class interface protection

In order to prevent indiscriminate access to application data, the declaration of data members has been placed in the private section of the classes. Therefore, they are only accessible to other member functions of their corresponding class. This restriction of access to the private data is a carefully considered principle of software engineering and is known as information hiding. On the other hand, the member functions have to be accessible to functions that are not members of their class. For that reason, they have been declared as public [43], [49].

5. Implementation of the relationships

In the design of the tool, there are four kinds of relationships: association, aggregation, inheritance and dependency relationships.

In the case of ***one-way association*** and ***aggregation*** relationships, since they are navigable in just one direction, the target classes become attributes of their corresponding classes in the relationship. For example, TIOA_Location as the target class of the "contains" relationship is an attribute in TIOA_States class. In the "has" relationship, each TIOA_Transition object has one TIOA_Location object as its

destination; therefore the TIOA_Transition class is an attribute in the TIOA_Location class.

In C++, the attributes mentioned above could be stored as pointers in the case of associations or as embedded objects or pointers in the case of aggregations. However, since the tool is assumed to be implemented in Java, they will be stored as references because in Java, objects are always references variables [49].

Also, if the multiplicity of the association or the aggregation is many, collection objects are used to store the attribute, i.e. list, vectors, and maps.

Dependency is another important relationship to show that the NFSM can't be built if the Grid Automaton can't be constructed and the Grid Automaton can't be constructed if the TIOA is not stored. Therefore, these dependencies serve to alert us to the fact that whenever we change the TIOA information, the Grid Automaton will be changed and so will NFSM. To implement the dependency relationship, it must be in the signatures of the methods as a type of a parameter. For instance, the "Sample" method in Grid_States class must have TIOA_States as a type of its parameter.

Another relationship, which is very powerful and useful tool to save a great deal of redundant effort, is the inheritance relationship. Objects in the classes NFSM_Transition and Grid_Transition inherit the attributes of class Transition (their parent) while having additional unique attributes of their own. The inheritance relationship is always one-to-one (excluding the representation of multiple inheritance relationships). In Java, inheritance is specified by using the keyword *extends* [43].

4.4.2 Evaluating our Design (A check list).

The evaluation points are [50]:

- *Most important criteria: correctness.*
  - Address and solve all issues in the problem definitions.
  - Solve the given problem within a specified constrains.
  - Design is complete: all classes and methods are needed, none is omitted.
- *Simplicity.*
  - Can any operations, data members, parameters be removed?
  - Can any parameters and return type or any operations be simplified?
  - Are the operations and their parameters lists logically consistent?
- *Cohesion and Coupling.*
  - Everything in a class is directed at single purpose. Split the class if it serves more than one purpose. Merge classes if they serve the same purpose.
  - Minimize dependencies on other classes.
    - All relationships among classes should be based on operations, not on data elements.
    - Minimize the number of the operations one class provides for another class. That is to simplify our interface.
- *Information hiding.*
  - Conceal the implementation details within a particular class or method.
  - Hides the potential changes to a particular class or method.
  - Data elements should be local (not visible from outside).

➢ Separate methods into those only accessible by other classes and those used locally (as a part of implementation).

➢ Try to identify heuristic classes.

❖ Portions of the design those are not well understood.

❖ Portions of the design that are machine or operating system specific (subject to change) or

❖ Portions of the design that are to be extended in the future

- *Error Handling.*

➢ A large percentage of the codes of most systems need to handle special cases and errors, e.g. Error messages are displayed when a wrong format of TIOA file is written.

➢ It is hard to add error handling to a system once it has been designed and implemented.

➢ Error handling should be part of the initial problem statement and should be important design criteria in considering alternative designs.

4.5 Scenario of Execution

For the testers to be able to use the tool, they can either run it on UNIX operating system (Figure 40) or on Windows (Figure 41 &42). It is important to follow the following steps by order.

On UNIX operating System:

1. Log into UNIX operating system.
2. All files should be placed in the same directory.
3. Enter the TIOA information to the file called "input". Note that the "input" file has no extension. For example if you rename it to input.txt, then the tool generates an error message. For writing TIOA description, testers must follow the format mentioned in Chapter 3.

4. Change to the directory that you have placed the files. Write in the command line: java TIOA_States to run the tool and have the results. The test cases' results of the small multimedia system are shown in Figure 40.

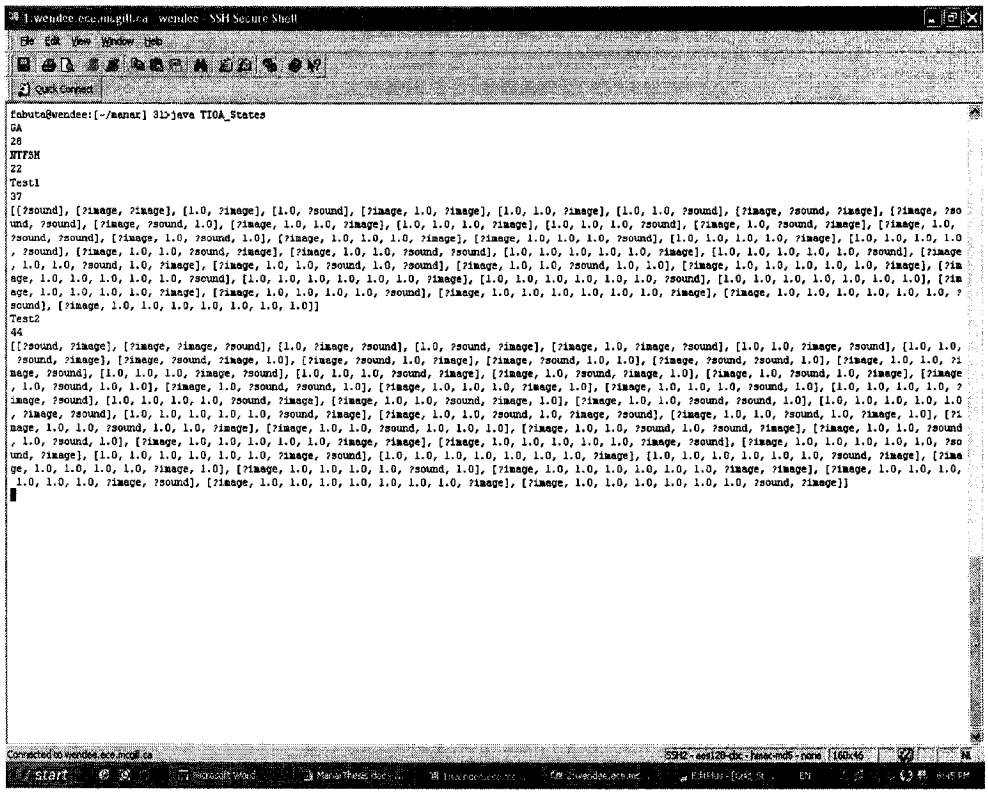

Figure 40: Running the tool (1)

5. You may have the results in a file by writing: (java TIOA_States >> outputfilename) in the command line.

Windows:

1. All files should be placed in "C:/manar" directory.

2. Enter the TIOA information to the file called "input". Note that the "input" file has no extension. For writing TIOA description, testers must follow the format mentioned in Chapter 3.

3. Open the file applet.html.

4. Click the required button for any results. For example, by clicking "Grid Automaton" button, the Grid Automaton information will be displayed in the text area. The Grid Automaton information is the Grid Automaton states with their transitions' information. See Figure 41 for the Grid Automaton

information of the small telephone system and Figure 42 for its Q set information.

Last thing to be mentioned is that the user must follow the specific file format in writing the TIOA description so that the tool can read the file correctly.

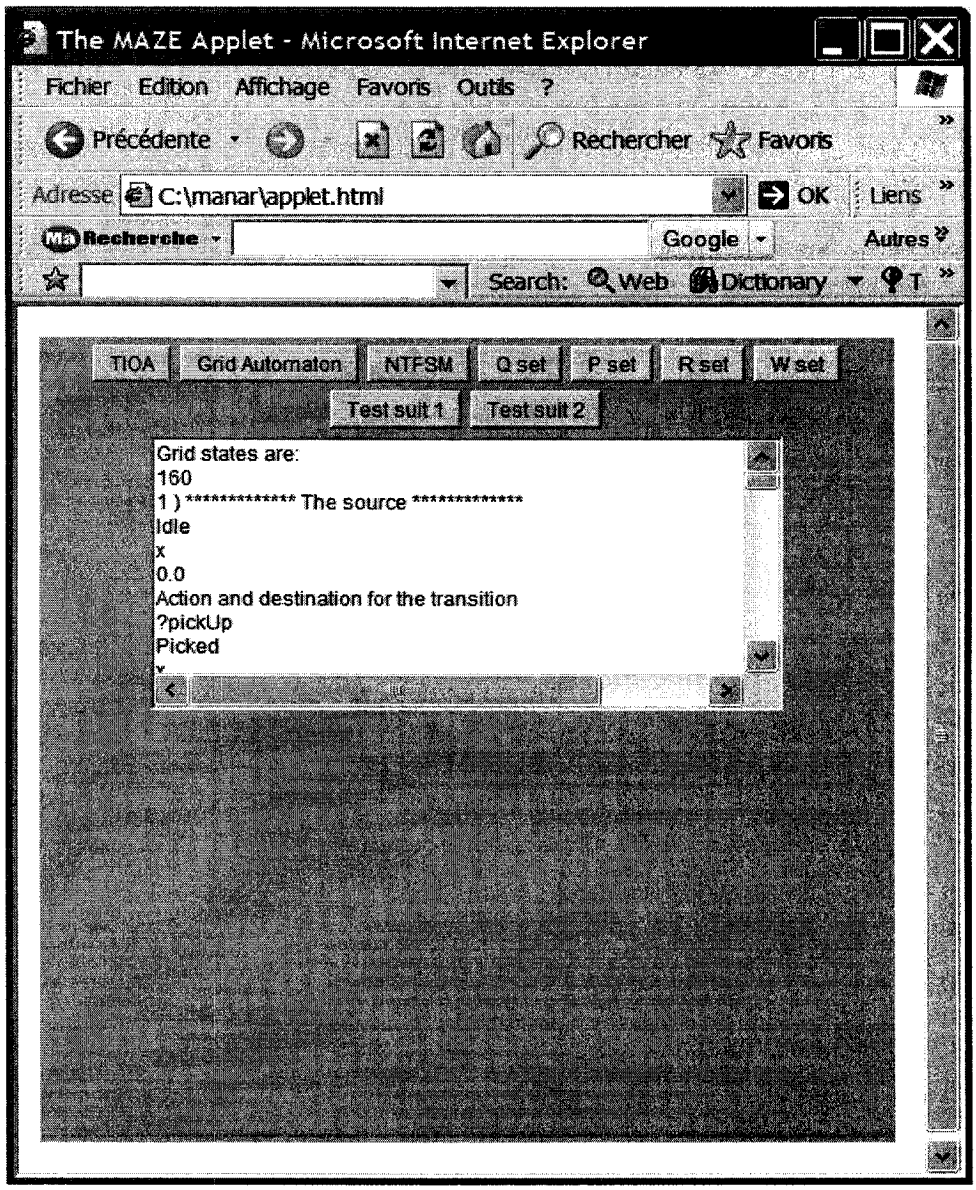

Figure 41: Running the tool (2)

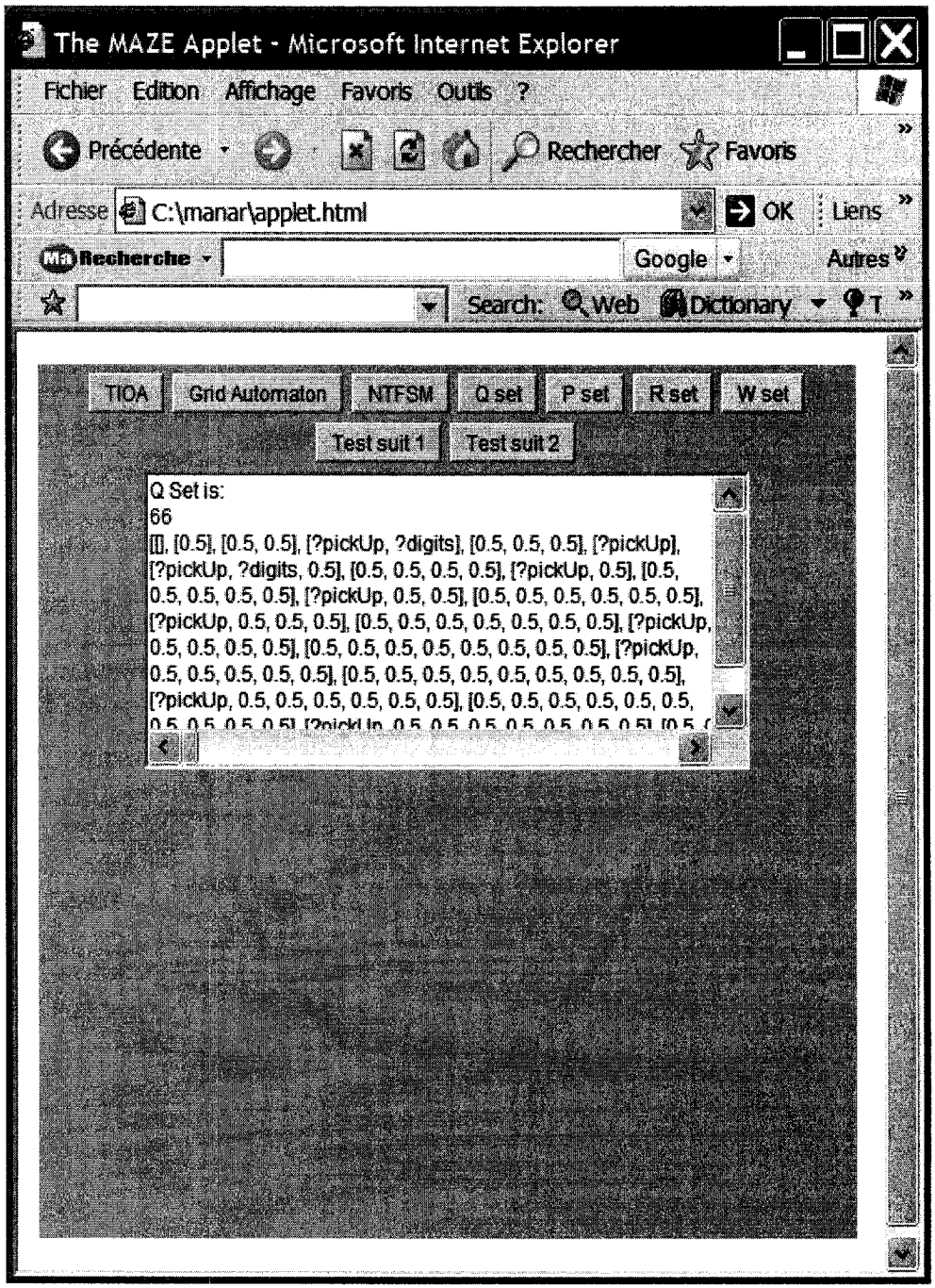

Figure 42: Running the tool (3)

4.6 Conclusion

In Chapter 4, major design issues are introduced and solved. The tool has been implemented in Java language and that is because of many reasons that were stated earlier in this chapter. JavaCC has been used as a parser of our developed tool and the grammar of TIOA specification has been also explained. The grammar is following the same format that was introduced in Chapter 3. The tool has a simple user interface and we showed how to use the tool by giving a scenario of its execution. Finally, we validate our design by a checklist method.

In Chapter 5, we run many examples using our tool. We also validate some of the results to ensure that the generated test cases are correct. We conclude with analyzing the results.

CHAPTER V: CASE STUDIES AND EVALUATION

Many examples are applied to our tool and it successfully detects many possible faults especially with the real time systems that have number of clocks (3 to 5) with medium time constraints. Four examples are described in this chapter with their results: number of Grid Automaton states, number of NFSM states, number of the test cases of the test suite one and two and the tool time response. The examples are applied according to different granularities. That is because the granularity of 1/ (2+n) may construct a very huge Grid Automaton with its information overhead. The tester can choose the granularity, which can be 1 or the common factor between the bounds of the time constraints. It is to be noted that the bigger the granularity is, the less the fault coverage is.

5.1 Hypothetical Telephone System

The telephone system [51] is responsible of establishing the connection between two parties after receiving the five digits as input from the first party. Figure 43 illustrates TIOA description of the timing constrains for such system. It has five states and the initial state is "Idle". When the user picks up the phone, the telephone system outputs the dial tone and initializes the clock to be equal to zero. As a result, it reaches the "Dial Tone" State. The time from picking up the phone to the dial tone signal can be any delay in the range [0, 1]. The total time to dial the five digits is any delay in the range [0, 15]; otherwise the behavioral constraints are violated and not considered. Moreover, the maximum delay between the dialing and the "connect" output is in the range [0, 10], where the "Connect" state is reached and the connection is established. One clock is used to evaluate the signal, the dialing time, and the connecting time. It

is noted that the TIOA for the telephone system is deterministic in order to trace the test cases. The results of the telephone system are shown in table 15. The machine that we have run the tool on is Intel P4 2.0GHz, 512 MB RAM, Windows XP home edition.

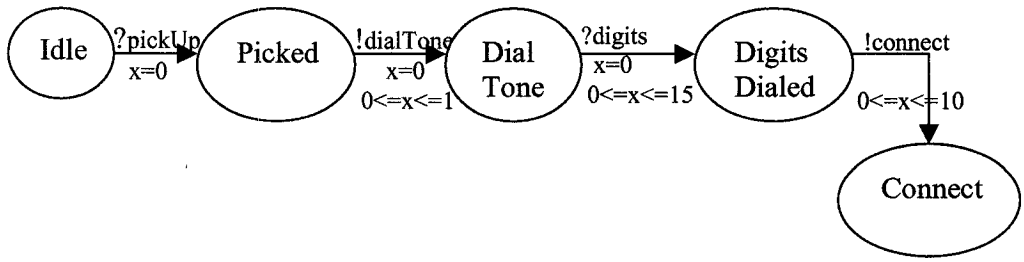

Figure 43: Telephone system (1)

Table 15: Telephone system results (1)

Granularity	GRID AUTOMATON States	Symbolic States	Transitions	Edges	Time
½	160	97	133	211	Real: 7m7523s
1	85	51	71	112	Real: 0m09.90s
5	25	14	22	31	Real: 0.844s

The same telephone system may have other dialing-time requirements. For example, the time between each successive pair of digits can be in delay in the range [0, 5]. Beside that, there is an error message if the total time to dial the ten digits is greater than 19 seconds. In such system, TIOA description differs from Figure 43. To represent the ten digits when they are dialed, Figure 44 shows the behavior of the system in terms of 14 states with their transitions. The results are discussed in table 16.

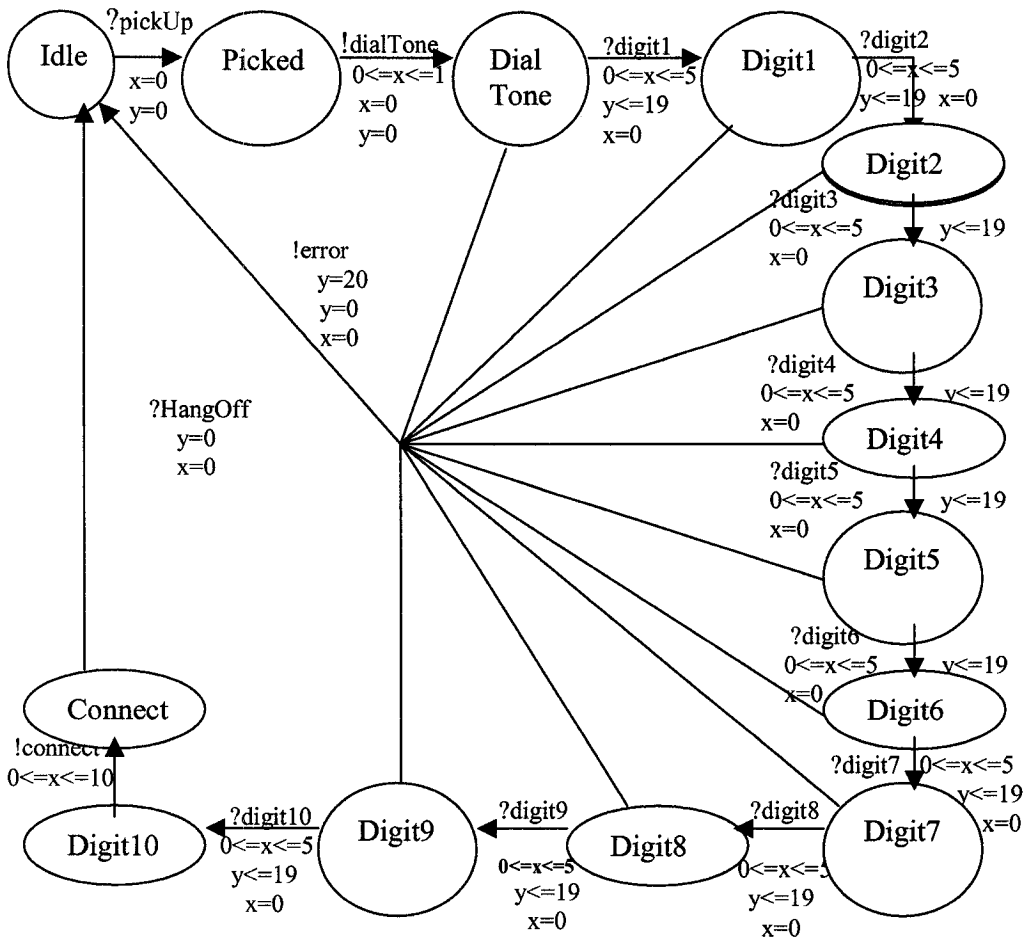

Figure 44: Telephone system (2)

Table 16: Telephone system results (2)

Granularity	Grid Subdivision	Number of Symbolic States	Number of Regions	Number of Edges	Time
1	2012	1250	12793	31814	Real: 643m54s

5.2 Small Multimedia System

The TIOA presented in Figure 45 shows the behavior of the small Multimedia System [52]. When the system is in the state "S1", the system is waiting for an image as input. When the image arrives, the system resets two clocks to zero and it moves to state

"S2" waiting for the sound as a second input (?sound) at most 2 seconds after receiving the first input (?image). If it receives a sound before the deadline, an acknowledgment will be sent at most 5 seconds; otherwise, it will discard the received inputs. At the end, after sending the acknowledgment, the clocks are again reset as an output behavior. Table 17 shows the system's results.

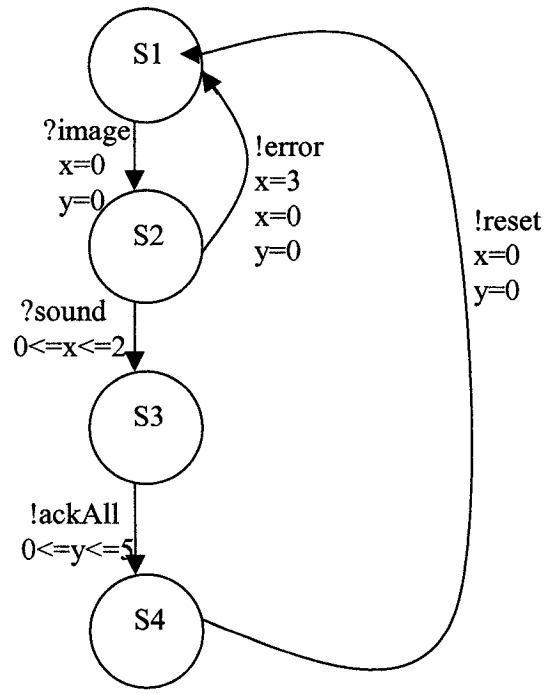

Figure 45: Multimedia system

Table 17: Multimedia system results

Granularity	Grid Automata States	Minimized STSG's States	Optimized Net States	Optimized Net Transitions	Real Time
¼	88	74	109	140	Real: 3m0317s
1	28	22	37	48	Real: 0m4.10s

5.3 Media Synchronization Protocol

The media synchronization protocol [53] is a protocol, which is responsible of synchronizing the real time continuous media such as video stream when the transfer rate quickly changes. Figure 46 shows the behavior of that protocol. The states: "S0", "S1", "S4" and "S5" try to send data continuously at the fixed rate to the states: "S2" and "S6". Note that a, b, d, x, and y are constant parameters specified by designers where:

1. The network propagation delay of sending a data may vary between the minimum "x" and the maximum "y".
2. "d" denotes the maximum time necessary for preparation to play a video.
3. The decoding time of the received data such as MPEG encoded data may vary between the minimum "a" and maximum "b".

First, S0 sends to S2 a video packet with a timestamp representing its sending time. S2 starts to receive the data at time x1 and finishes to receive the data at time x2 where (x1+x <= x2 <= x1+y). Next, the system outputs a signal, which instructs to start playing the video packet before x2+d. Then, it outputs a signal, which instructs to finish playing the video packet after the necessary duration to finish playing [a, b]. After that, S6 carries out the above work continuously. Moreover, it synchronizes the playing rate with the sending state. To do so, it sets the objective starting time for playing the next video packet to time x3+w, where x3 is the starting time for playing the previous video packet and w is a difference between two timestamps of the most recent video packet and its previous one. If the actual completion time of receiving data is earlier than the objective one (x6<x3+w), the system waits until the objective time and then outputs the signal to start playing. Once again, TIOA description in Figure 46 is deterministic, which has 5 clocks.

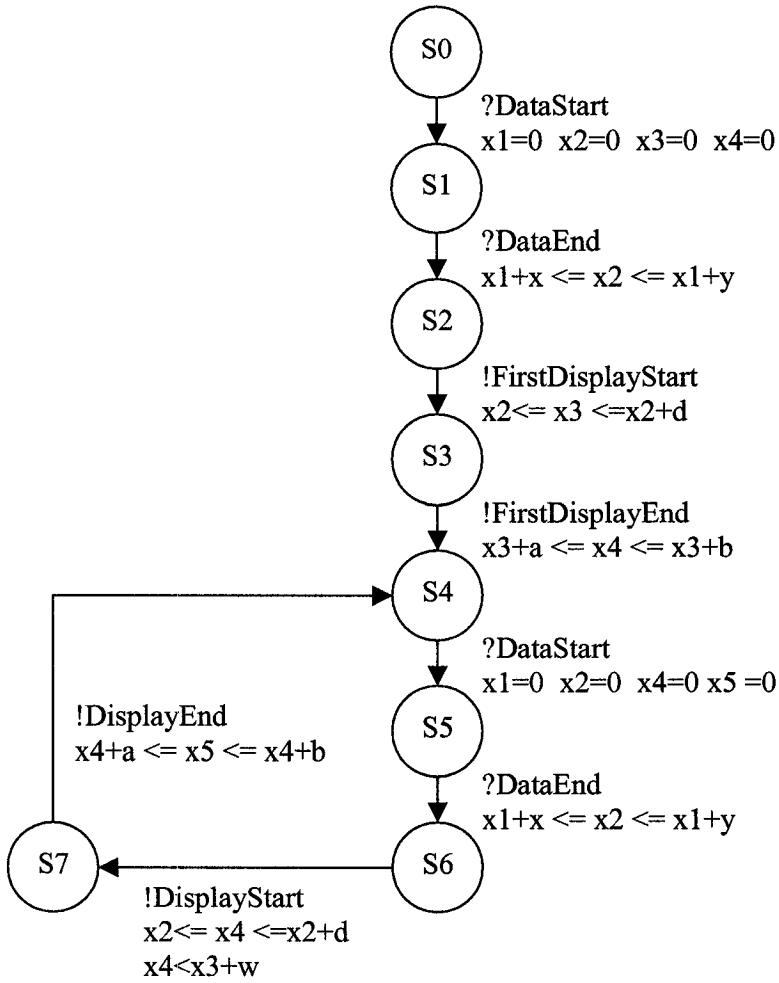

Figure 46: Media synchronization protocol (1)

To execute the media synchronization protocol example and have the test suite, the constant parameters must be specified with some values. If it is assumed that the network propagation delay is not high where x= 5 and y = 15, d=10 and the decoding time in the range from a=5 to b =30. One case can be the following:

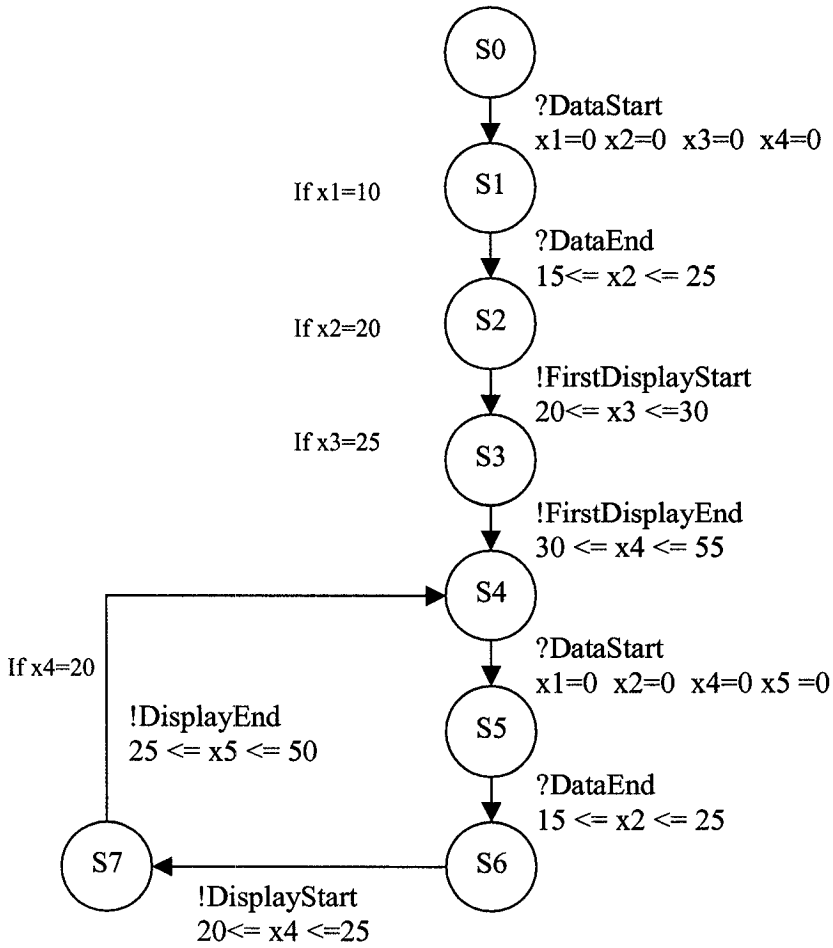

Figure 47: Media synchronization protocol (2)

Then the results for such protocol are as following:

Table 18: Results of the media synchronization protocol (1)

Granularity	Grid Automaton States	Minimized States	Optimized States	Symbolic States	Processing Time
1	3082	1967	2926	2271	Real:320m1365s
5	232	153	210	225	Real: 9.062s

Another assumption can be when the network propagation delay is higher than before where x= 60 and y = 100, d=10 and the decoding time in the range from a=10 to b =40. Figure 48 shows TIOA description of such protocol:

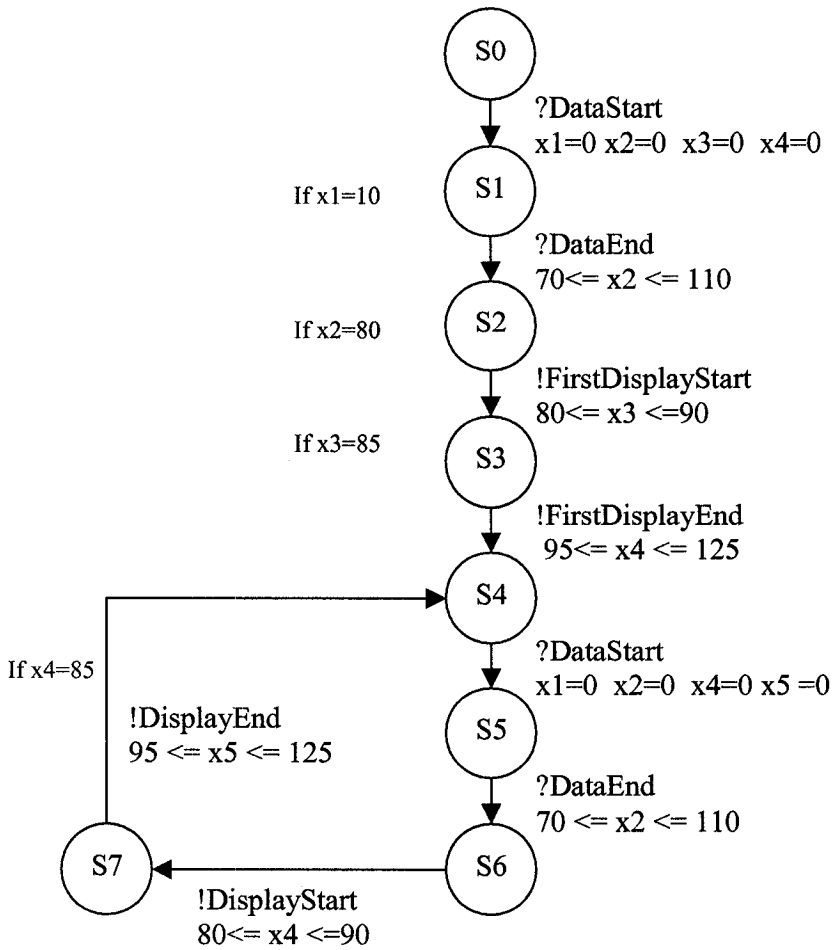

Figure 48: Media synchronization protocol (3)

Then the results for such protocol are as following:

Table 19: Results of the media synchronization protocol (2)

Granularity	Grid Automaton States	Minimized NFSM States	Optimized Test Cases	Optimized Test Cases	Optimized Test Cases
1	11, 795	9340	Real: More than one day and there is no test cases.		
5	649	499	861	902	Real: 2m8143s

5.4 Railroad Crossing System

In the railroad system [54], [55], [56] shown in Figure 49, more than one train can cross a gate simultaneously, through multiple parallel tracks. According to the train's destination, the train can independently choose the gate it will cross. Each gate is controlled by one controller, which must be active all the time to close and open the gate for the trains. Two kinds of objects are determined in such system: environmental objects and system object. The environmental objects include: trains and gates. The system object is the controller. The trains are interacting with the controllers through sensors (many –to-many relationships). The controllers are communicating with the gates through actuators (one-to-one relationship). Two properties must be satisfied in the railroad system: the safety property and the liveness property. The safety property is that the gate remains closed whenever there is a train inside a crossing. The liveness property is that the gate eventually reopens when the last train leaves the crossing.

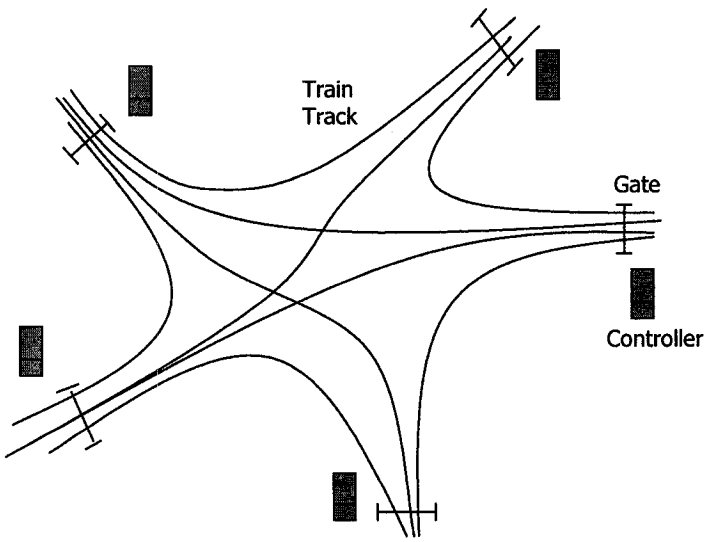

Figure 49: Railroad crossing system

The timing assumptions for the railroad crossing system are as following:

- A train (Figure 50) enters the crossing within an interval of [60, 120] seconds after sending a (!Near) message to a controller informing that it is approaching. A train then informs the controller (by sending (!Exit) message) that it is leaving the crossing within 180 seconds of sending the approaching message.

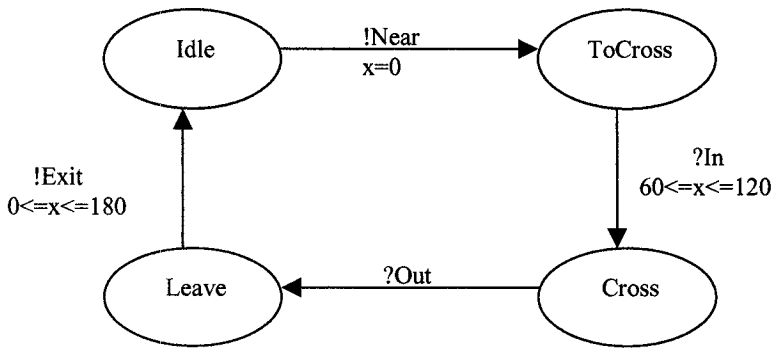

Figure 50: Train

- A gate (Figure 51) closes within 60 seconds after receiving the instruction (?Lower) from a controller. A gate then triggers the opening event (!Up) within an interval of 0 to 60 seconds after receiving the instruction (?Raise) from a controller.

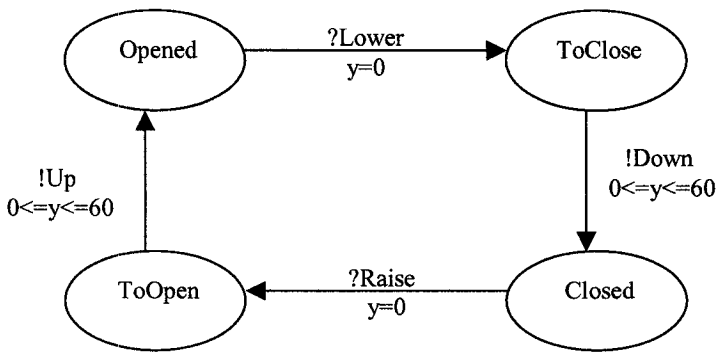

Figure 51: Gate

- A controller (Figure 52) instructs the gate to close within 120 seconds after receiving an approaching message (?Near) from the first train entering the crossing. A controller then instructs a gate to open within 120 seconds after receiving an exiting message (?Exit) from the last train leaving the crossing.

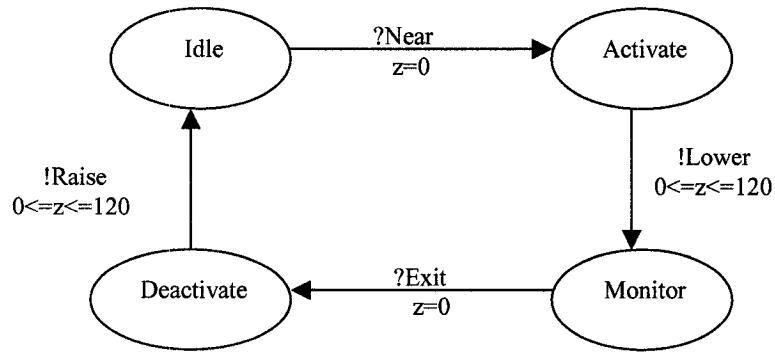

Figure 52: Controller

It is noted that the railroad crossing system is composed of three deterministic TIOA descriptions, which have the shared events between them. The system uses three clocks to keep a track of the timing constraints. The results obtained from the tool for each TIOA are mentioned in the following three tables:

Train:

Table 20: Train results

Granularity	Grid Automaton States	Untimed Reachable States	Untimed Edges	Timed States	Time
½	1208	1084	1929	3247	Real: 5h22m58.6s
1	608	544	969	1627	Real: 1h48m29.2s
2	308	274	489	817	Real: 1m41s
5	128	112	201	331	Real: 3.781s
60	18	13	25	34	Real: 0.813s

Gate:

Table 21: Gate results

Granularity	Grid Automaton States	Minimized NFSM States	Optimized Test Cases	[illegible]	[illegible]
½	488	486	729	1335	Real: 14m40s
1	248	246	369	675	Real: 1m2s
2	128	126	186	345	Real: 3.734s
5	56	54	81	147	Real: 2.906s
60	12	10	15	26	Real: 0.485s

Controller:

Table 22: Controller results

Granularity	Grid Automaton States	Minimized NFSM States	Optimized Test Cases	[illegible]	[illegible]
½	968	966	1449	2655	Real: 1h4m045s
1	488	486	729	1335	Real: 8m48s
2	248	246	369	675	Real: 1m125s
5	104	102	153	279	Real: 3.047s
60	16	14	21	37	Real: 0.828s

5.5 Results Validation

Before analyzing the results we have obtained from the tool, the validation step is necessary to be done in order to ensure that the generated test cases from our tool are correct. The purpose of that test result validation is to determine whether the trace of interactions observed during a test satisfies the requirements of the reference specification. This result, sometimes called verdict, which is not easy to be obtained in general; one sometimes refers to an Oracle to provide an answer. As it is shown in Figure 53, we have the test cases as a result of our tool and the starting point for the conformance testing is the definition of the conformance relation between an IUT and the specification. Does the implementation confirm to its specification? As it was

explained earlier in chapters 1 and 2, the environment behaves according to the specification reference and by observing the implementation reactions (observed outputs) to the applied inputs; we can compare them with the expected outputs of the specification within the allowed time interval. Accordingly, in Oracle the test cases are first applied to the TIOA specification to obtain the expected output. They are then applied to the IUT to derive the observed output. If the observed output is equal to the expected output then the TIOA specification and the IUT are conformed, otherwise the fault is detected in the implementation of the system.

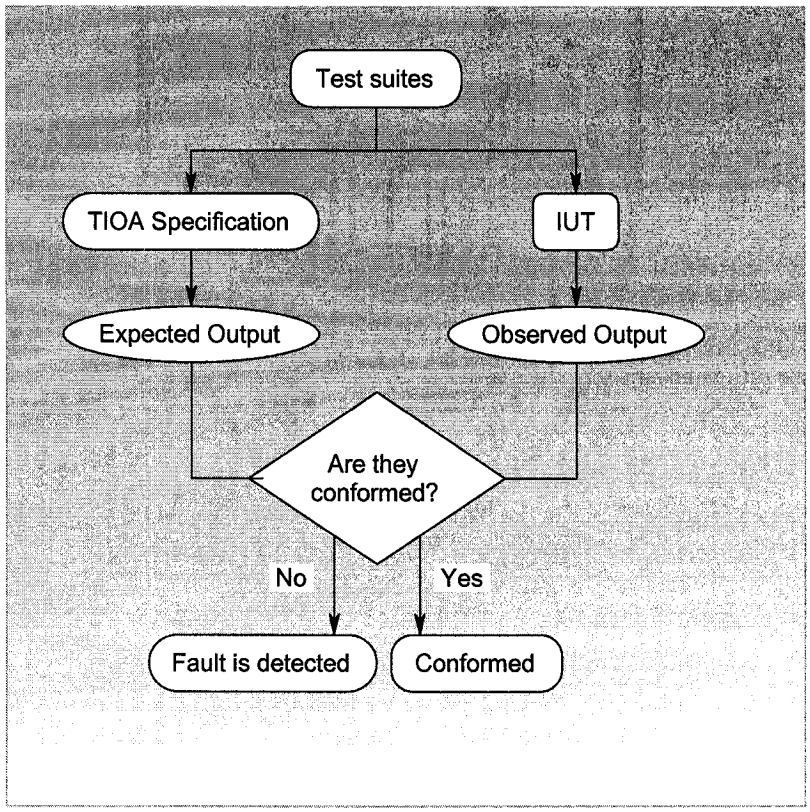

Figure 53: Oracle methodology

Let's validate the obtained test cases from the multimedia system (see Figure 54 and 55). The TIOA specification of the multimedia system was introduced in 5.2 section.

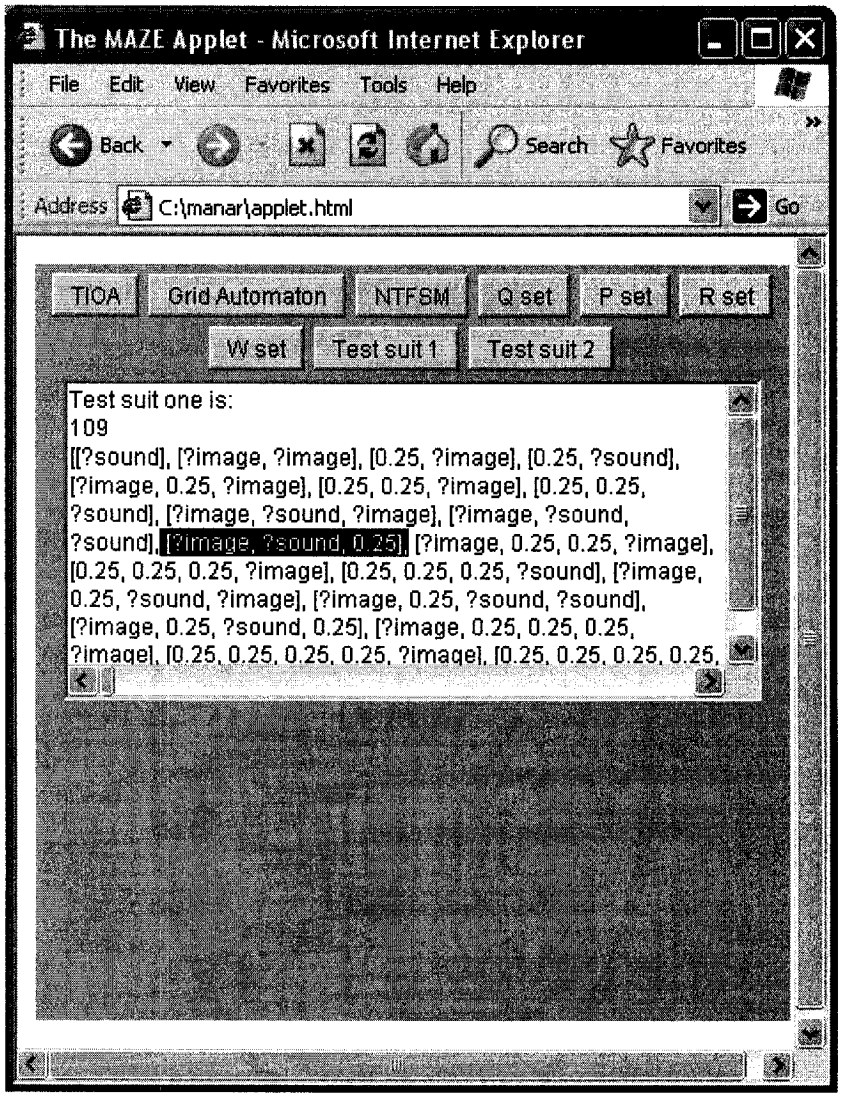

Figure 54: Test suite one of the multimedia system

Figures 56 and 57 show two faulty implementations. The implementation in Figure 56 has an output fault in location S2. It doesn't respond with the output *!error* after the application of the input sequences {*?image.?sound.* ¼ }. This input sequences is obtained from the test suite one (Figure 54, the shaded test case), which produces the expected output {*!error&resetXY.-.-* } that is not conformed to the observed output {*-.-.!ackAll* }. Therefore, the output fault is detected in the faulty implementation (Figure 56).

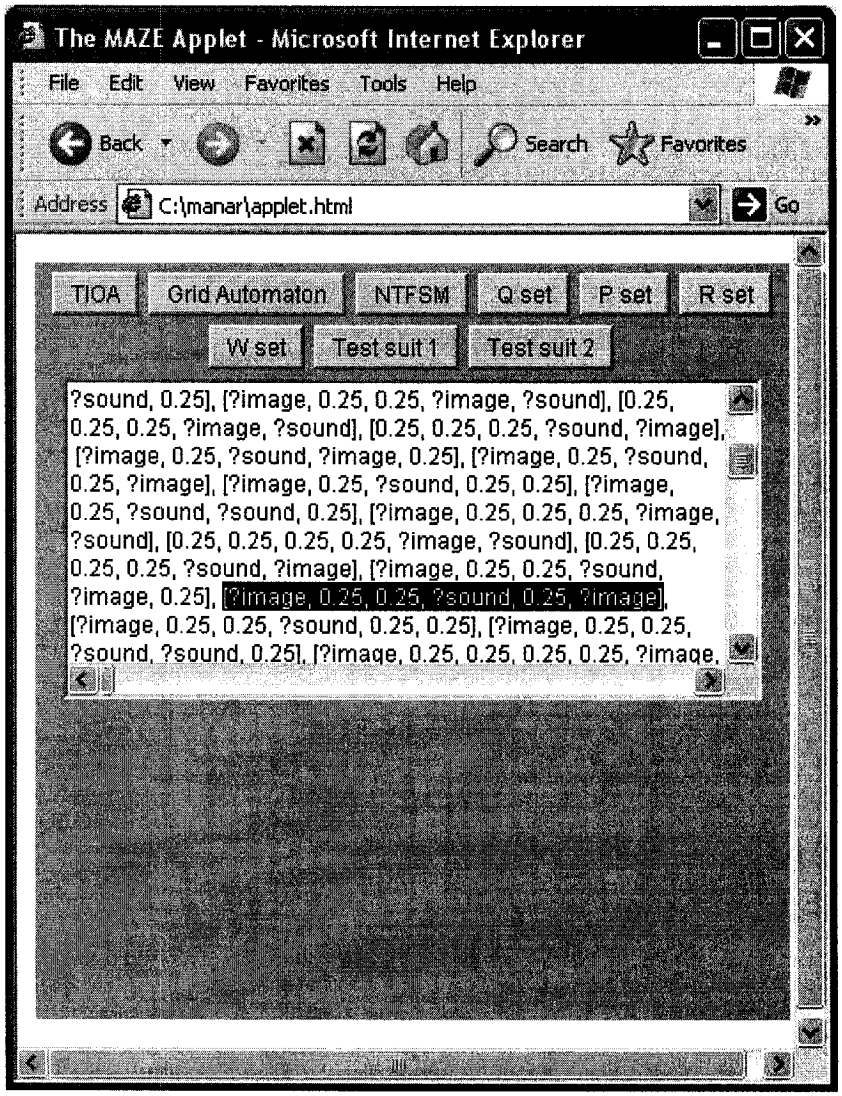

Figure 55: Test suite two of the multimedia system

Moreover, the implementation in Figure 57 has a transfer fault in state (S4, x= ½, y= ½). Indeed, after the sequence {*?image. ¼ . ¼ .?sound. ¼ .?image*} , which is obtained from the test suite 2 (Figure 55, the shaded test case) , the expected output that is derived from the TIOA specification is {-.-.-.*!ackAll.!resetXY.!error&resetXY*} which is not conformed to the observed output {-.-.-.*!ackAll. !resetXY.*- } that is derived from the implementation. Therefore, the fault is detected where the faulty implementation enters the state (S3, 0, 0) instead of the state (S1, 0, 0).

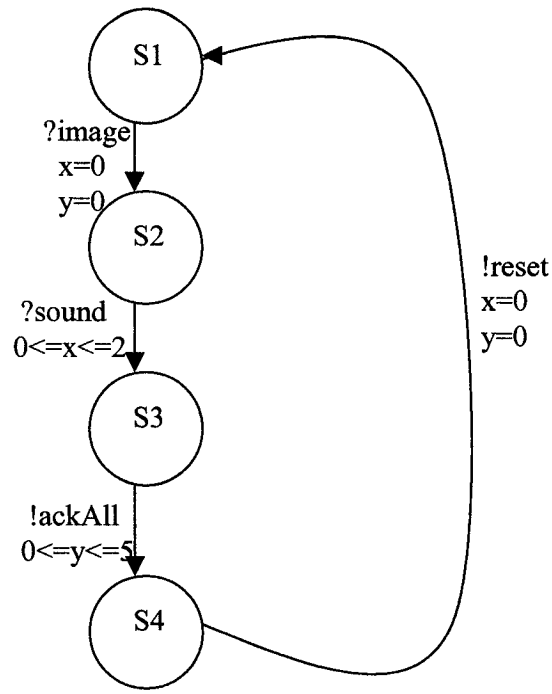

Figure 56: Multimedia system with output fault

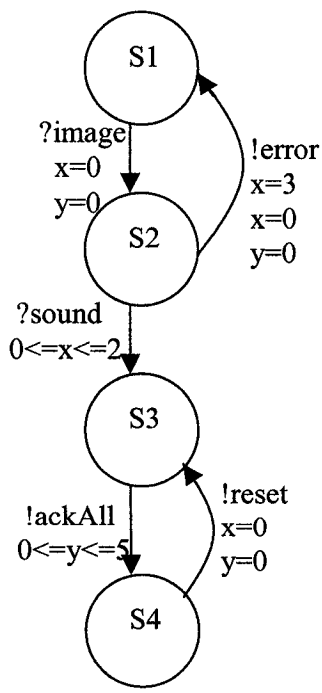

Figure 57: Multimedia system with transfer fault

Other faults that can also be detected using the generated test cases of our tool are the time constraint widening and restriction faults. The implementation of Figure 58 enlarges the time constraint of the transition from S2 to S3 on input *?sound* when the value of clock x is less than 3. Contrary to the specification, this implementation accepts the input *?sound* even when the value of clock x is between 2 and 3. The input sequence: {?image. ¼ . ¼. ¼ . ¼ . ¼ . ¼ . ¼ . ¼ . ¼ .?sound} , which is obtained from the test suite one, detects such fault type. The expected output is {-.-.-.-.-.-.-.-.-.-.-} that is not conformed to the observed output {-.-.-.-.-.-.-.-.-.-.*!ackAll*}. The faulty implementation enters the state (S4, 2 ¼, 2 ¼) instead of the state (S2, ∞, ∞). The same test case can detect the time constraint restriction fault if the implementation restricts the same time constraint to make the value of clock x is less than 1.

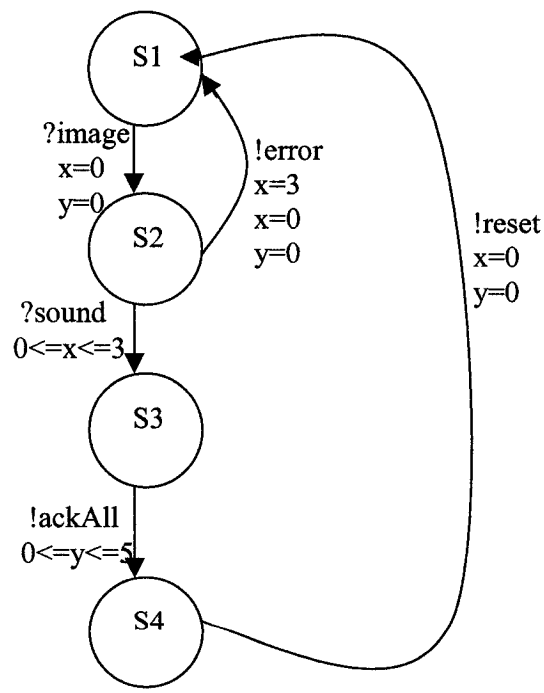

Figure 58: Multimedia system with time constraint widening faults

Finally, the test cases that are generated from our tool are also able to detect the reset clock faults. We also consider the TIOA of the multimedia system in 5.2 section. The implementation in Figure 59 doesn't reset the clock x with the transition from S1 to S2. The faults related to a reset of a clock change the ordering between clocks in the implementation. In our case, the implementation in Figure 59 changes the ordering between clocks x and y since the specification requires that the clock x must be not greater than the clock y in location S2. It also decreases the number of the reachable states. The input sequence: {¼ .?image} , which is obtained from the test suite one, detects such fault type. The expected output is {-.*!error&resetXY*} that is not conformed to the observed output {-.*!error&resetY*}. The faulty implementation enters the state (S1, ¼, 0) instead of the state (S1, 0, 0).

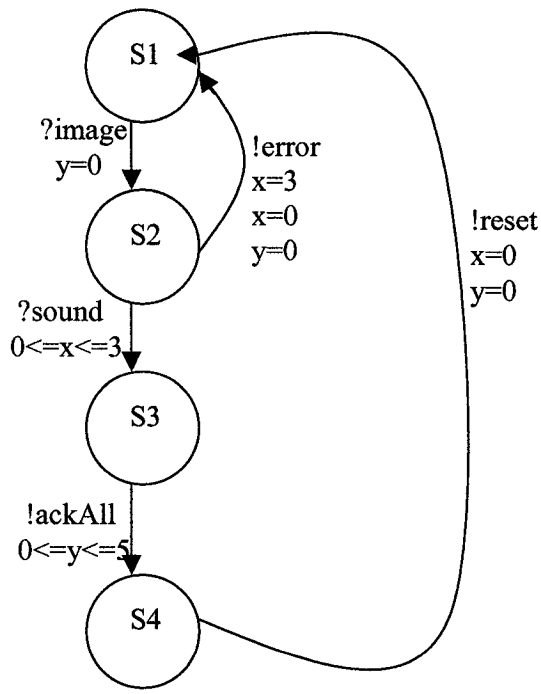

Figure 59: Multimedia system with reset clock fault

5.6 Results Analysis

By analyzing the obtained results in the tables introduced in the previous sections in this chapter, we noticed the following:

- Even if the sampling algorithm is exponential on the number of the clocks and the constants used as bounds in the time constraint, the tool remains scalable with the real time systems of most two clocks, medium integer constants and the granularity of 1 and it generates the test cases in less than 20 minutes.

- If the tool is applied with the real time systems of three clocks to five clocks, which is the rare case, and large integer constants, then the test cases are generated and executed in more than 5 hours. Otherwise, it may take days to have the test cases for the real time systems with the large number of clocks and the integer constants.

- For any input transition, there exists at least one test case that covers the time space of the transition.

- The tool is also used to derive the timed test cases for the real time systems specified as a set of communication TIOA [57]. This serves testing the embedded real time systems. All the parts of the complete or the partial product of the system are composed to be tested. Refer to the example of railroad crossing system in 5.4 section.

- The most time execution is spent on generating Q set and the test suite two. For example, when Dijkstra algorithm [47] is applied for NFSM whose states' number is larger than 1000 states, then each state is checked with the rest of the 999 states to find the shortest distance for that state.

- It is to be noted that when the granularity of 1 is used instead of 1/(2+n), then the test cases are approximately reduced by (1-(1/(2+n))). The bigger the granularity is chosen, the fewer the test cases are generated and the less time is needed to generate them.

- Whenever there is a clock reset, there are more generated states. For example, the telephone system (2) has a reset action for "x" clock in many transitions. This leads to new states with different locations where x is equal to zero and y has different values.

5.7 Conclusion

In this chapter, we run the tool on many examples such as the telephone system, the multimedia system, the synchronization protocol and the railroad crossing system. The tool detects successfully many possible faults as we have seen in the multimedia system, using the oracle methodology to validate its generated test cases. Finally, we have analyzed the results and made our conclusions on the developed methodology.

CHAPTER VI: CONCLUSION

Real-time systems are more challenging to test than non-real-time systems. A number of design decisions affect the testability of the real-time system such as the parser, choosing an algorithm... etc. Our tool is efficient enough to produce the required test cases by applying a method, called Generalized Wp-method. In order to generate the test cases, the tool is divided into four parts. Firstly, the parser, which parse TIOA file and match the real time system information to its corresponding data structure in TIOA_States part, which is the second part of the tool. Thirdly, Grid_States is the part that is responsible for sampling TIOA in order to construct the Grid Automaton according to certain granularity determined by the tester. The granularity can be 1/2, 1/n+2, 1 or a common factor between TIOA time constraints. Finally, to have a good coverage of the potential faults in a timed system, NFSM_States is the part that takes care of transforming the Grid Automaton into NFSM and minimizing NFSM. It then generates the test cases based on the state characterization sets and optimizes them. We have run the tool to various examples with different sizes of states, clocks and time constrains. The tool successfully generates the timed test cases many examples. Even if the sampling algorithm is exponential on the number of the clocks and the time constraints, choosing the proper granularity and minimizing NFSM are factors make the tool scalable. The tool is also powerful to derive the timed test cases for a set of communicating real time systems.

Future Work

To improve our software tool there are some features that can be implemented in the future such as the transformation of NFSM into ONFSM. The tool can also be expanded to deal with non-deterministic TIOAs. Moreover, there are many design

patterns [39], which can be used to increase the efficiency of our tool such as MVC architecture, etc. It is also a good idea to implement the tool in C++ to increase the tool performance in terms of time response.

The tool must also have user interface, which achieves the security requirements. This user interface must have the activity logging and the management access. Certain people must use the tool and each user has limit rights of using the tool.

Lessons learned

In summary, we have also learned new things while developing our tool to test the real time systems. We are excited by this modest achievement. The integration of our tool can leverage the real-time systems and provide us with new and fertile research ground while addressing the information explosion, a very real and important task. In addition, documenting the tool [50] is a very important task especially that the readers refer to the documentation rather than the code. The clearer the documentation is, the more benefits come out of it in terms of product improvement and maintenance. We referred to IEEE standards [33] to make sure that many specification and design issues are covered. Moreover, UML is used to describe the structural and the behavioral model of the tool. Rational Rose was the tool for drawing the UML diagrams. We have also learned more concepts about the graph theory, compiler theory, applets and Java Language. It is nice experience to deal with many examples and generate their test cases through our tool. It has been deeply analyzed the importance of the tool results to the real time systems.

REFERENCES

[1] M. Grindal and B. Lindström, "Challenges in Testing Real-Time Systems.," presented at In 10th International Conference on Software Testing Analysis and Review (EuroSTAR'02), Edinburgh, Scotland., 2002.

[2] Jim Turley, "Embedded Processors by the Numbers," *Embedded Systems Programming*, vol. 12, 1999.

[3] W. Schütz, *The Testability of Distributed Systems*: Kluwer Academic Publishers, 1993.

[4] H. Kopetz and P. Veríssimo, "Real Time and Dependability Concepts," in *DistributedSystems*, S. Mullender, Ed.: Addison-Wesley, 1993.

[5] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*, Third Adition ed, 2001.

[6] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek, "Timed Wp-Method: Testing Real-Time Systems," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, pp. 1023-1038, 2002.

[7] IEEE, "BS 7925-," British Standardisation Institute 1998.

[8] D. Clarke and I. Lee, "Automatic generation of tests for timing constraints from requirements," presented at 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97), Newport Beach, CA, USA, 1997.

[9] Elaine J. Weyuker and Sandra Rapps, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering (TSE)*, vol. 11, pp. 367-375, 1985.

[10] Mingyu Yao, "On the development of conformance test suites in view of their fault coverage," in *Computer Science*. Montreal: Université de Montréal, 1996.

[11] O. Charles, "Application des Hypothèses deTest à une Définition de la Couverture," in *Computer Science*. Nancy: Université Henri Poincaré - Nancy 1, 1997.

[12] Son T. Vuong and Jadranka Alilovic-Curgus, "On Test Coverage Metrics for Communication Protocols," presented at Protocol Test Systems 1991, 1991.

[13] P.G. Frankl, R. G. Hamlet, B. Littlewood, and L. Stringini, "Evaluating Testing Methods byDelivered Reliability," *IEEE Transactions on Software Engineering*, vol. 24, 1998.

[14] B. Beizer, *Software Testing Techniques, second edition*: Van Nostrand Reinhold, 1990.

[15] Rajeev Alur and David Dill, "A Theory of Timed Automata," *Theoretical Computed Science*, vol. 162, pp. 183-225, 1994.

[16] Rajeev Alur and David Dill, "Automata For Modeling Real-Time Systems," presented at Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, 1990.

[17] T. S. Chow., "Testing software design modeled by finite-state machines," presented at IEEE Transactions on Software Engineering, SE-4(3), 1978.

[18] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machine," 1996.

[19] G.V.Bochmann, A.Das, R.Dssouli, M.Dubuc, A.Ghedamsi, and G.Luo, "Fault Models in Testing." North-Holland, 1991.

[20] S.Natio and M.Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours," presented at Proceedings of Fault Tolerant Computer Systems, 1981.

[21] J.Edmonds and E.L.Johnson, *Matching Euler Tours and the Chinese Postman*, 1973.

[22] R. Nahm, "Conformance Testing Based on Formal Description Techniques and Message Sequence Charts," University of Bern, 1995.

[23] G. Gonenc, "A Method for the Design of Fault Detection Experiment," *IEEE Transactions on Computers*, pp. C-19: 551-558, 1970.

[24] S.T.Vuong, W.W.L.Chan, and M.R.Ito, "The UIOv Method for Protocol Test Sequence Generation," presented at Proceeding of the 2nd International Workshop Protocol Test System, Berlin, Germany, 1989.

[25] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and A. Ghedamsi., "Test Selection Based on Finite State Models.," presented at IEEE Trans. Software Eng. 17(6), 1991.

[26] Gang Luo, G. von Bochmann, and A. Petrenko, "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method," *IEEE Transactions on Software Engineering*, vol. 20, pp. pp. 149-162, 1994.

[27] D.Mandrioli, S.Morasca, and A.Morzenti, "Generating Test Cases for Real-Time Systems from Logic Specifications," *ACM Transactions on Computer Systems*, pp. 13(4): 365-398, 1995.

[28] F.Liu, "Test Generation Based on an FSM Model with Timers and Counters," University of montreal, 1993.

[29] K.Springintveld, F.Vaadranger, and P.Dargenio, "Testing Timed Automata," *Theoretical Computed Science*, pp. 254:225-257, 2001.

[30] R.Milner, *Communication and Concurrency*: Prentice-Hall International, 1989.

[31] K.Cerans, "Algorithmic Problems in Analysis of Real Time System Specifications," in *Computer Science*: University of Latvia, 1992.

[32] R. Alur and C.Dill, "A Theory of Timed Automaton," *Theoretical Computed Science*, pp. 126:183-235, 1994.

[33] I. Standard., "Software Requirements Specifications," 1998.

[34] A. En-Nouaary, "Génération de cas de test pour les systèmes temps réel modélisés par des automates à entrées sorties temporisées," in *Département d'informatique et de recherche opérationnelle Faculté des arts et des sciences*. Montreal: University of Montreal, 2001, pp. Univérsité de Montréal.

[35] R. Waldura, "Dijkstra's Shortest Path Algorithm in Java," vol. 2004: Renaud Waldura, 2002.

[36] Clifford and A. Shaffer, "A Practical Introduction to Data Structures and Algorithm Analysis," 1996.

[37] M. Main, *Data Structures and Other Objects Using Java*: Addison Wesley Longman, 1999.

[38] I. Sun Microsystems, "Model-View-Controller," 2004.

[39] R. H. Erich Gamma, Ralph Johnson and John Vlissides, *Design patterns Elements of Reusable Object-Oriented Software*: Addison Wesley Professional, 1994.

[40] I. Sun Microsystems, "Java BluePrints Model-View-Controller," 2002.

[41] Andrew Kitchen, "Programming Language Translation," 1994.

[42] A. V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*: Addison-Wesley, 1986.

[43] John Lewis and W. Loftus., *Java Software Solutions: Foundations of Program Design, Update*, 2nd ed: Addison-Wesley, 2002.

[44] John R. Levine, Tony Mason, and Doug Brown, *Lex & Yacc*, 1990.

[45] T. S. Norvell, "Introduction to JavaCC," U. O. B. COLUMBIA, Ed. Vancouver, 2003.

[46] R. Cox., "C++ v. Java," vol. 2004, 1998.

[47] Ravindra K. Ahuja, Thomas L.Magnant, and J. B. Orlin, "Network Flows: Theory, Algorithms and Applications," 1st ed, 1993.

[48] W. Mai and S. Ameri, "Software Design Methodologies," vol. 2004: Concordia Computer Science Department, 2002.

[49] James P.Cohoon and J. W. Davidson, *C++ Program Design: An Introduction to Programming and Object-Oriented Design*, 2nd edition ed: McGraw-Hill College, 1997.

[50] B. Appleton, "A Software Design Specification Template," vol. 2004, 1997.

[51] D. Clarke and I. Lee, "Automatic generation of tests for timing constraints from requirements," presented at 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97), Newport Beach, CA, 1997.

[52] H. Fouchal, M. Defoin-Platel, S. Bloch, and P. M. E. Petitjean, "Timed Automata Generation from Estelle Specifications," presented at ESTELL 98, Paris, 1998.

[53] T. Higashino, A. Nakata, K. Taniguchi, and A. R. Cavalli, "Generating test cases for a timed I/O automaton model," presented at In G. Csopaki, S. Dibuz, and K. Tarnay, eds, Proc. IFIP Int'l Work. Test. Communicat. Syst. (IWTCS), Budapest, Hungary, 1999.

[54] R. Cardell-Oliver, "Conformance Testing of Real-Time Systems against Timed Automata Specifications," 1999.

[55] Vangalur S. Alagar and K. Periyasamy, *Specification of Software Systems*: Springer Verlag, 1998.

[56] D. O. Ormandjieva, "System Requirements Specifications (COMP6481)," vol. 2003, 2003.

[57] A. En-Nouaary, F. Khendek, and R. Dssouli, "Testing Embedded Real-Time Systems," presented at 7th International Conference on Real-Time Computing Systems and Applications (RTCSA), Cheju Island, South Korea, 2000.

APPENDICES

Appendix A

Parser Structure

//Structure of Transition method

```
TIOA_Transition Transition() :
{
        Token t;
        TIOA_Transition myTransition;
        String Dest;
        String Action;
}
{
    < Desti >
    t = <Others>
    {Dest = t.image;}
    t = <Others>
    {Action = t.image;}
        {myTransition= new TIOA_Transition(Dest, Action, null , null);}
        {return myTransition;}
}
```

//Structure of constraint method

```
Vector constraint() :
{
        Token t;
    String name;
    String oper;
    int bound;
    Vector CONSTs;
    TIOA_Constraint constr ;
}
{
    {CONSTs =  new Vector();}
    < CONSTRAINT >
    (
        t = <Others>
        {name = t.image;}

        t = <Others>
        {oper = t.image;}

        t = <Others>
```

```
            {bound = Integer.parseInt(t.image);}

            {constr = new TIOA_Constraint(name, oper, bound);}
            {CONSTs.addElement(constr);}
        )+
        {return CONSTs;}
}
```

//Structure of reset method

```
Vector reset() :
{
        Token t;
        String CLK;
        Vector CLKs;
}
{
    CLKs = new Vector();}
    <RESET>
    (
        t = <Others>
        {CLK = t.image;}
        {CLKs.addElement(CLK);}
    )+
    {return CLKs;}
}
```

Appendix B

Activity Diagrams for a Concrete Context

Sampling Activity Diagram

The activity diagrams below are used here to understand and model the operations of all main algorithms across a concrete context of the following TIOA:

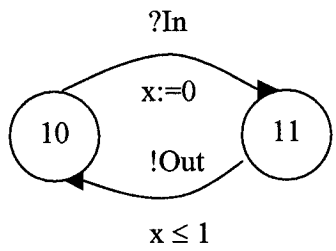

Figure 60: TIOA with one clock

The flow of activity of Sampling Algorithm (Figure 61):

When the Parser stores the information of the file to its corresponding data structure of TIOA, TIOA is then sampled to Grid Automaton. The following activities are taken into considerations:

1. The number of the clocks is obtained, which is one.
2. The granularity is calculated, which is ½.
3. The first grid state is the first TIOA state "l0". In addition, It has the clock information to identify the grid state: a clock name x and a clock value of 0.
4. Adding that state to reachable_states.
5. Extracting a state from (reachable_states - handled_states), it is again the initial state (l0, 0).
6. Adding the transition information to (l0, 0). Two kinds of transitions are added, the input transition and the delay transition.
7. The destinations of the transitions are added to reachable_states which are (l1, 0) and (l0, ½).

8. Adding (l0, 0) to handled_states, since we are done with that state.

9. Again, extracting another state from the set (reachable_states - handled_states), which is (l1, 0).

10. Adding the transition information to (l1, 0).

11. The destinations of the transitions are added to reachable_states, which are (l0, 0), (l1, ½). However, (l1, ½) is only added to reachable_states since (l0, 0) is already there.

12. In the same manner, the other states and their transitions will be added to Grid_States.

One thing to be mentioned is that the state (l0, 1) will not have an input transition since the bound of the time constraint is one as it is shown in the Figure 60. Therefore, its delay transition has a destination of (l0, infinity). (l0, infinity) has also a delay transition to itself. Finally, the algorithm stops when set (reachable_states - handled_states) is empty.

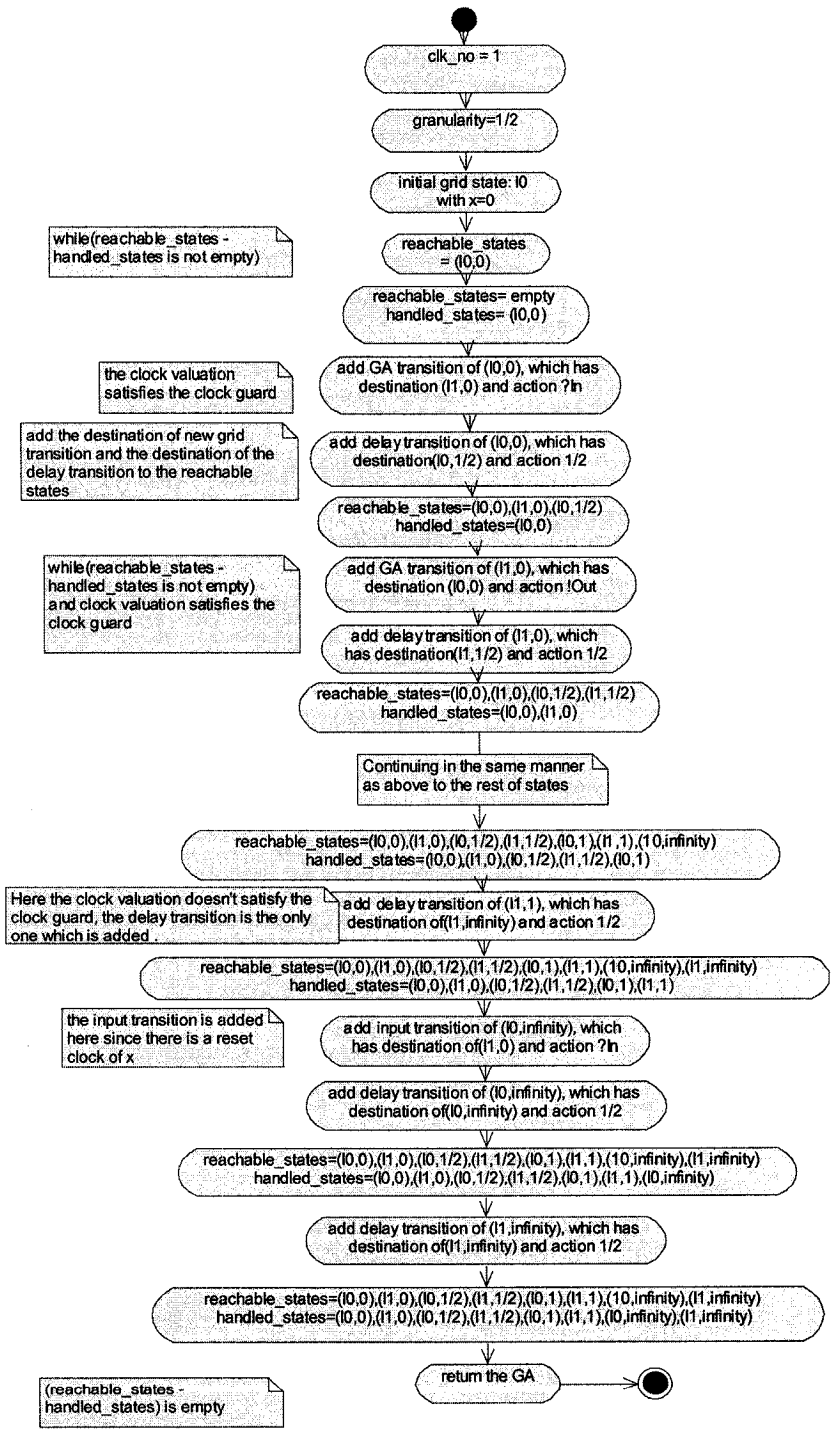

Figure 61: Sampling activity diagram

Transforming Activity Diagram

The flow of activity of Transforming Algorithm (Figure 62):

When Grid_States is constructed, then it is transformed to NFSM_States object. The transformation actions are as following:

1. Getting the first grid state from Grid_States object and make a new NFSM state that has the same location and the clock information of that grid state.

2. For each transition of that grid state, if it has an input or a delay action and if its destination has a transition that have an output action, which starts with !, then make a new NFSM transition. In our case, (l0, 0) has ?In as input action and it has a destination (l1,0) whose transition has the output action !Out to state (l0,0), then the new NFSM transition is from (l0,0) to (1,0). Another NFSM transition is from (l0, 0) to (l1, 0), whose action is ?In|-.

3. Another new NFSM transition, which also must be added, is the delay transition from (l0.0) to (1, ½), which has ½ as input action and without any output action.

4. For the rest of the grid states, their corresponding NFSM states are created with the same location and clock information. As well as their input transitions and the delay transitions are added to them.

5. After finishing with the input and the delay transitions, the action of the reset clock must be checked for each state whether or not it has a destination with a clock value of zero. In case of (l0, 0) NFSM state, it has two destinations with x = 0, therefore, they must be added to the output action in the related transitions. The same is done for the states (l0, ½), (l0, 1) and (l0, infinifty). Each of them has two destinations with x=0, which are (l0, 0) and (l1, 0).

6. Finally, to make NFSM_States object more specified, each state must have all inputs, which starts with ?. Checking each NFSM state, (l1, 0), (l1, ½), (l1,1), and (l1,infinity) are the states, which must add NFSM transitions whose action is ?In|-. These transitions goes their states.

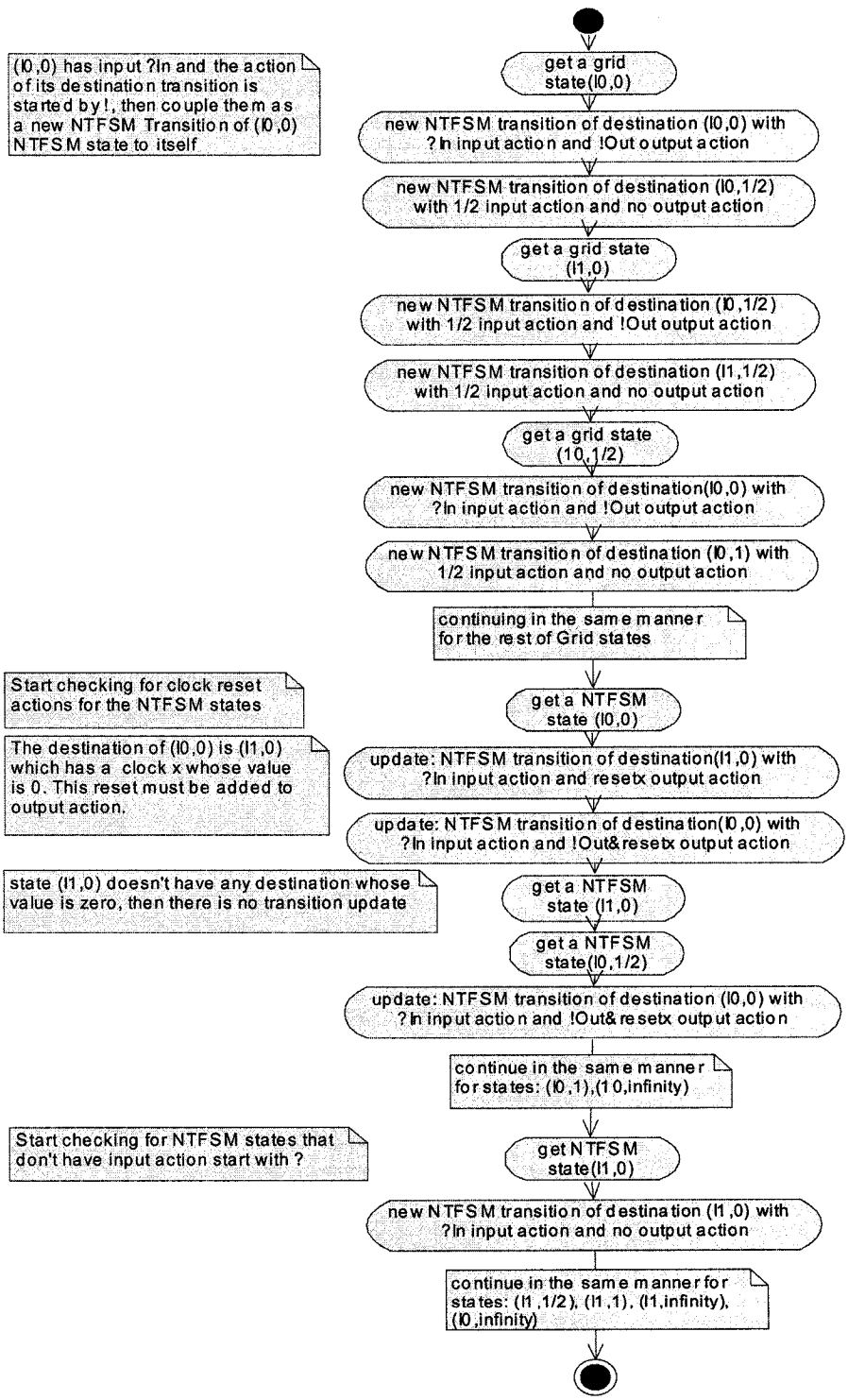

Figure 62: Transformation activity diagram

Minimization Activity Diagram

The flow of activity of Minimization Algorithm (Figure 63):

When Grid_States is transformed to NFSM_States, then NFSM_States must be minimized before generating the test cases considering the following activities:

1. NFSM_States must be checked whether or not it has equivalent states. We go through each two states, which have different locations but the same clock information, and check if they have the same transition information. If yes, then the equivalent state is removed after attaching its input to its next state (destination). In our case, [(l0,0) ,(l1,0)], [(l0, ½) ,(l1, ½)], [(l0,1) ,(l1,1)] and [(l0,infinity) ,(l1,infinity)] are the states, which must be checked for the equivalence.

2. States: (l0, 0) and (l1, 0) are not equivalent since their transitions have different information from each other.

3. Repeating the same procedure for the rest of the states, we noticed that the NFSM_States object is already minimized.

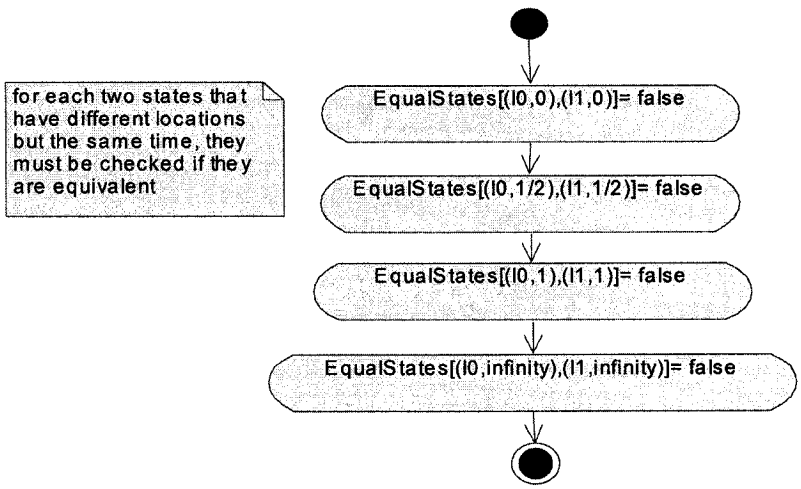

Figure 63: Minimization activity diagram

Test Cases Activity Diagram

The flow of activity of Test Suite One and Test Suite Two Algorithms (Figure 64):

Once the NFSM_States is minimized, then it is ready to obtain the test cases from it as follows:

1. To generate the first test suite, Q set must be obtained by calling its function and W set must be obtained by calling its function. As a result of concatenating Q with W, the test suite one is generated.

2. Next is generating the second test suite, R set is obtained by calling its function. For each element in R, find its corresponding reachable state from the initial state of NFSM by applying the same sequence of inputs of R element and then calculate Wi for that reachable state. As a result of concatenating each R element with its Wi, the test suite two is generated. In our case, the first element of R is [?In, ?In] and its corresponding reachable states are (l0,0) and (l1,0). When [?In, ?In] is applied from the initial state of NFSM which is (l0,0), first input ?In leads to two reachable states (l0,0) and (l1,0) and applying the second input ?In to these two states leads to the same reachable states (l0,0) and (l1,0), which are number 1 and 5 from the states. Now calculating W1 of state (l0,0), which is ?In, and calculate W5 of (l1,0), which is ?In. By concatenating R element [?In, ?In] with W1 and W5 [?In], the result is [?In, ?In, ?In]. Repeating the same procedure to the rest of R elements to obtain the second test suite.

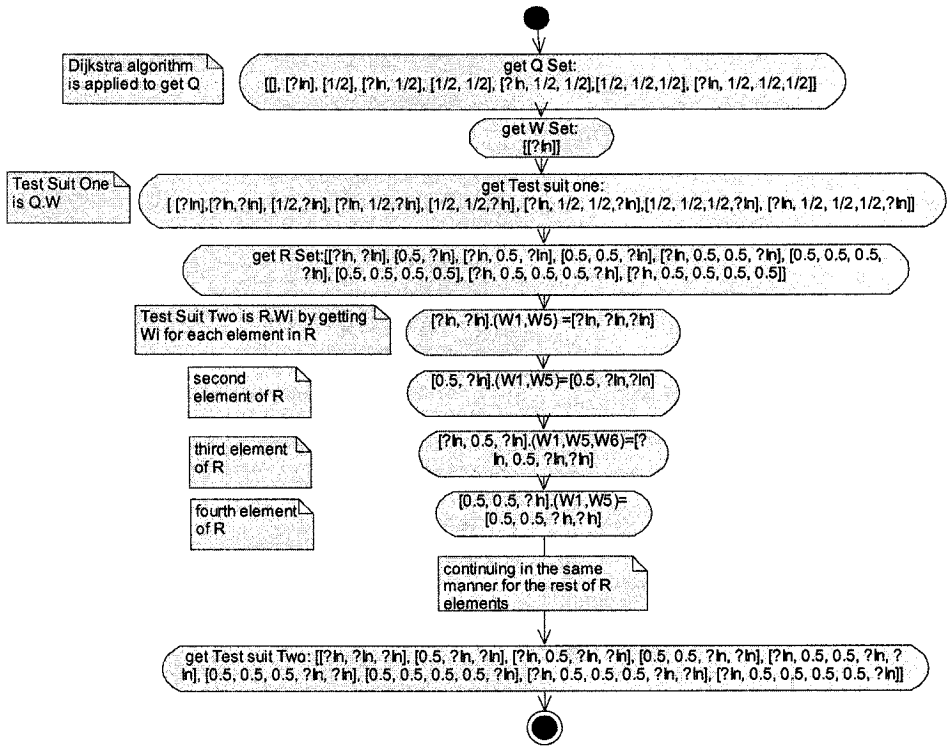

Figure 64: Test cases activity diagram

Q Activity Diagram

The flow of activity of Q Algorithm (Figure 65):

How to get Q set, which is necessary for the test suite one, is explained in the following actions:

1. Initially, the settledNodes contains the state (l0,0) and the unsettledNodes contains the state (l0, ½). When extracting the minimum distance from unsettledNodes, (l0, ½) is chosen since it is the only one in unsettledNodes. Then putting it in settledNodes.

2. Relaxing its neighbors means that each adjacent state to (l0, ½) is included to unsettledNodes. They are (l0, 1) and (l1, 0). Note that (l0, 0) is a neighbor but it is not added to the set since it is already settled. Then set the predecessor of (l0, ½) to be mapped to (l0, 0) and set the shortestDistances to be mapped to 1 instead of infinitiy.

3. Getting the next minimum distance from unsettledNodes which is (l0,1). It can also be (l1, 0) since they have the same value of distance which is infinity.

4. Relaxing its neighbors after adding it to settledNodes set. Its neighbor, which is not settled yet, is (l0,1½). Set the predecessor of (l0,1) to be mapped to (10, ½) and set its shortestDistances to be mapped to 2 instead of infinity.

5. After we are done with all states. We go again over NFSM states. For each NFSM state, we get its predecessor in order to get the input action and store it in a vector. Let's take the case of (l0,1) state. Its predecessor is (l0, ½) and therefore the input from (l0, ½) to (l0,1) is stored in the vector, which is ½ . We have to get now the predecessor of (l0, ½) which is (l0,0). The input from (l0,0) to (l0, ½) is also stored in the same vector, which is ½ . The vector contains the input sequence [½ , ½] and reversing it means having the same vector. Finally, this vector is stored in Q vector to represent one element of Q set.

6. Repeating the same procedure to the rest of NFSM states.

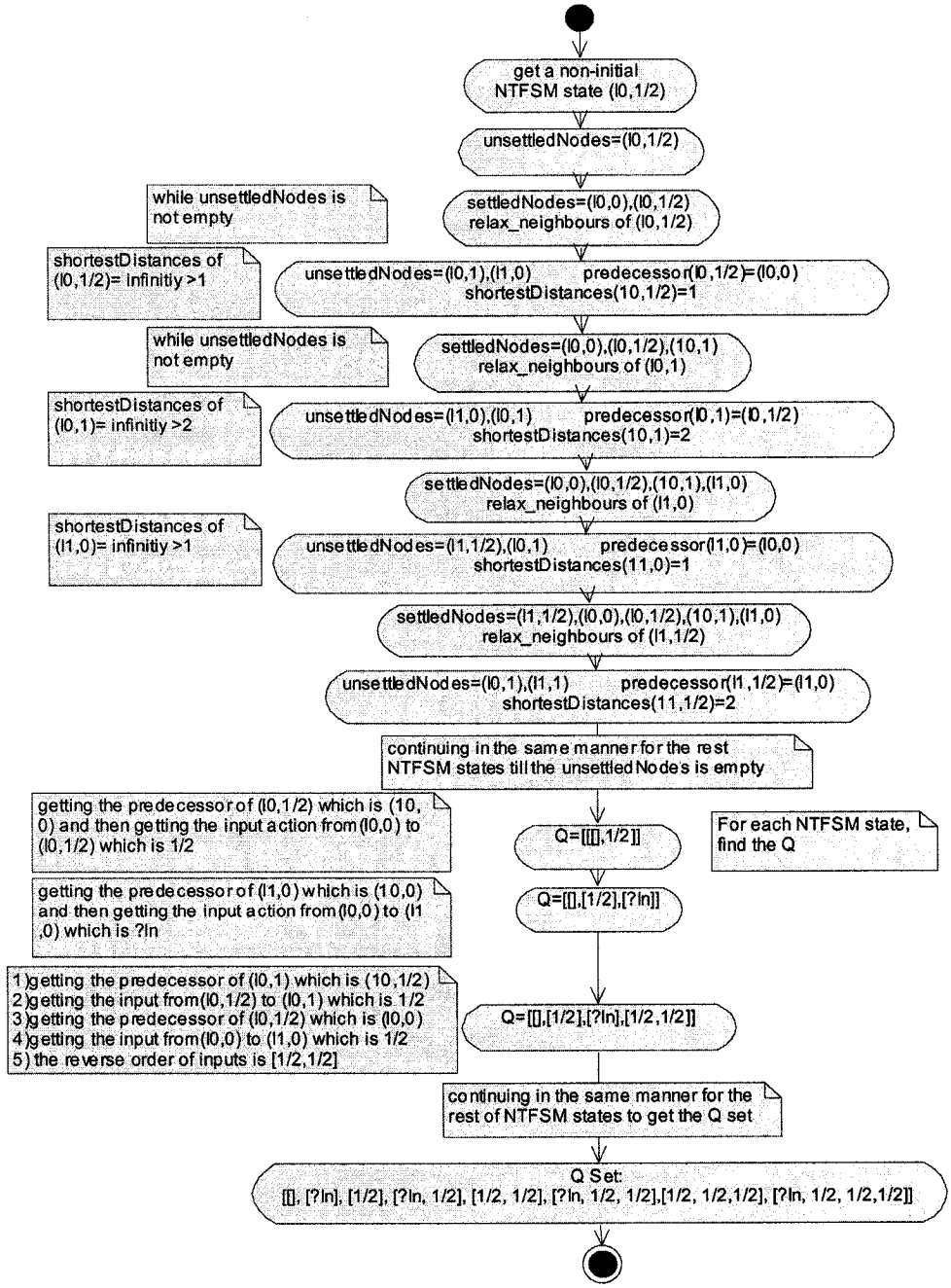

Figure 65: Q activity diagram

Wi Activity Diagram

The flow of activity of Wi Algorithm (Figure 66):

How to get Wi set, which is necessary for obtaining the test suite two and W set, is explained in the following actions:

1. For each NFSM state, we go through the input vector to find an input element that is applied to other states, which have different locations but the same clock information, gives different output. Starting with an input: ?In, (l0,0) and (l1,0) two states, which have different locations but the same time: x=0, have different outputs: !Out&resets and !Out.

2. Repeating the same procedure to the rest of NFSM states. It is noticed that the input: ?In, when it is applied to the rest of states, is sufficient to give the different output.

In case if the input vector doesn't have any element that gives different outputs, then the input vector is concatenated with itself to have new input elements: [[?In, ?In], [?In, ½],[½ ,?In], [½ , ½]]. Again the same method is applied to the states in order to find one input element that gives different outputs.

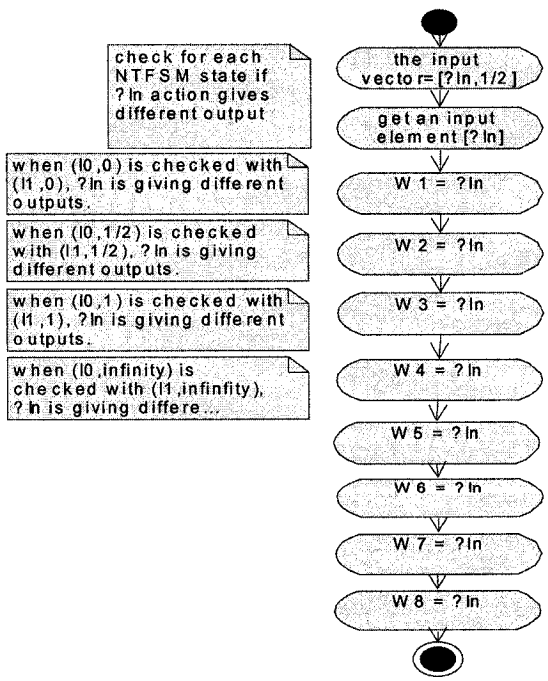

Figure 66: Wi activity diagram