TTRTS: A Tool for Testing Real Time Systems

Abdeslam En-Nouaary and Manar Abu Talib

Electrical and Computer Engineering Department
Concordia University, Montreal, Canada
ennouaar@ece.concordia.ca
m_abutal@ece.concordia.ca

Abstract: Real-time systems (RTS for short) are simply those systems whose behavior is time dependent. We encounter many of such systems in our daily life (e.g., airplanes traffic control systems, patient monitoring systems, etc.). The correctness of RTS is very critical because a functional misbehavior may have harmful consequences on both human lives and environment. Testing is one of the formal techniques that can be used to ensure the correctness of RTS. In this paper, we present a tool that generates timed test cases for RTS modeled as Timed Input Output Automaton (TIOA). The proposed tool is based on state identification technique and the test cases generated by the tool have full fault coverage. We applied our tool on many examples with different sizes. Such examples and their corresponding results are presented in this paper.

Key words: Real-Time Systems, Formal Methods, Conformance Testing, Timed Automata.

1 Introduction

The testing phase is one of the most time, effort and cost consuming phases in software development cycle. It consists of executing the implementation of a system with the intention to discover errors. This is done in three steps. First, test cases are generated from the specification of the system, which we suppose to be correct. Then, we apply the generated test cases on the implementation and we observe its reactions. Finally, we analyze the test results. If, for each test case, the observed outputs are the same as the expected ones then the implementation is said to be conformed to its specification; otherwise the implementation is said to be faulty and the diagnosis process starts to locate and fix the faults.

Many aspects should be considered when testing a system. First of all, we should consider a fault model. The fault model is an important parameter in testing because it allows us to know the potential faults that might exist in the implementation under test (IUT for short) and helps us to generate test cases that could discover those faults. Another important concept in testing is the fault coverage. It refers to the power of a test cases generation method to detect all the possible faults included in the fault model. This can be used to compare test cases generation methods. Indeed, a method A is said to be more powerful than a method B if A has a better fault coverage than B. However, for a more accurate comparison, other parameters should be taken into account such as the length of the test suite. The last important concept to be mentioned is the conformance relation between the implementation being tested and its specification. The conformance between an implementation and its specification means that the implementation is equivalent to the specification with respect to some mathematical relations (Brinksma et al., 1986; G.V.Bochmann et al., 1991; Susumu Fujiwara et al., 1991).

Over the past three decades, many works have been done in testing (see for instance (Abdeslam En-Nouaary & al., 2002; Brinksma et al. 1986; Cardelli-Oliver 1999; Chow, 1978; D. Clarke & al., 1997; G.V.Bochmann et al. 1991; K.Springintveld & al., 2001; Susumu Fujiwara et al. 1991; T. Higashino & al., 1999)). Although testing principles developed for non-real-time systems are generally applicable to real-time systems, the consideration of the timing behavior of RTS complicates many issues. First of all, the fault model is not straightforward. Moreover, the development of efficient tests generation methods is difficult because the executability of test cases should be taken into account. Finally, the conformance relation has to deal with the density of the time domain, which is infinite.

In this paper, we propose a modular and scalable tool to generate timed test cases for real-time systems modeled as Timed Input Output Automata (TIOA for short). The proposed tool is based on state identification technique (Chow, 1978; G.V.Bochmann et al. 1991; Susumu Fujiwara et al. 1991) and consists of the following four components (Figure 1):
The parser is used to parse the TIOA specification given in a text file and match its information to a data structure. The TIOA file is written by the tester who is responsible of describing the desired RTS to be tested. The TIOA specification must be deterministic.

The sampler is used to sample the TIOA specification to come up with a subset of the system’s behavior that can be easily tested. This is done by using a suitable granularity (i.e., a special time delay) by which the system is allowed to go through all its timed conditions upon the progression of time. The resulting behavior is described by a finite automaton, called Grid Automaton (GA for short) (Abdeslam En-Nouaary et al. 2002; K.Springintveld et al. 2001).

The converter is responsible of transforming the GA into a Finite State Machine (FSM for short). The resulting FSM is also minimized to remove the equivalent states so that we can generate test cases from it based on state identification techniques.

Finally, the generator is used to generate test sequences by adapting the Generalized Wp-method (G.V.Bochmann et al. 1991). The generator also optimizes the test cases to be generated by removing those, which are prefixes of others.

The tool was mainly implemented based on the framework published in (Abdeslam En-Nouaary et al. 2002). However, some decisions and parameters, which were not discussed in (Abdeslam En-Nouaary et al. 2002), have been made when implementing the tool as well as some new algorithms such as the minimization of the automaton to be tested and the optimization of test cases to be generated.

The remainder of this paper is organized as follows. Section 2 gives an overview of the tool. Section 3 is devoted to some case studies and the results obtained by the application of the tool. Section 4 concludes the paper and presents some future works.

2. Tool Overview

As mentioned previously, our tool consists of four parts, namely: the parser, the sampler, the converter and the generator. Each of these parts has its own responsibilities and is relatively independent. Moreover, the implementation of the tool is done in Java (John Lewis & al., 2002) because of its numerous advantages.

In the rest of this section, we will give an overview of each of the components of the tool as well as the algorithms implemented by each component.

Figure 1. Tool’s Structure

2.1 The Parser

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 and break it down into pieces that are useful to the program (Aho & al., 1986; John R. Levine & al., 1990; Norvell 2003). Analyzers and parsers are usually used to accomplish this task. In our case, we use Java Compiler Compiler (JavaCC) to analyze and parse the textual specification of a real-time system prior to test cases generation. The specification of the RTS under test is given as Timed Input Output Automaton whose definition is as follows.

Definition 1: Timed Input Output Automaton

A Timed Input Output Automaton (TIOA) A is a tuple (IA, OA, LA, l0_A, CA, TA), where:
- IA is a finite set of input actions. Each input action is denoted by “!” followed by a label.
- OA is a finite set of output actions. Each output action is denoted by “?” followed by a label. Note IA ∩ OA = φ
- LA is a finite set of locations. The term “location” is chosen instead of the term “state” because the latter is used to define the operational semantics of TIOA.
- l0_A ∈ LA is the initial location.
- CA is a finite set of synchronous clocks set to zero in l0_A. We assume that time is dense, which means that the clocks values are real numbers.
- TA is the set of transitions. Each transition consists of a source location, an input or an output action, a clock guard that should hold in order to execute the transition, a set of clocks to be reset when the transition is executed, and a destination location. We assume that the transitions are instantaneous.

An example of TIOA is shown in Figure 2. Here, the TIOA has two locations l0 and l1, one clock x, and two transitions. The transition $l0 \xrightarrow{?a,x>1,c>0} l1$ executes from the location l0
on input ?in only if the value of the clock x is less or equal to 1. In that case, the automaton resets the clock x to zero and changes its location to l1.

![Diagram](image-url)

**Figure 2. An Example of TIOA**

JavaCC generates a lexical analyzer program that reads and analyzes the input source and converts its character strings into individual Token objects. In the case of a timed automaton, a token is a location, an input, an output, a clock, an operator, or a bound of a time constraint. The tokens are defined by the grammar rules set up in the .jj specification file. Therefore, the user must be careful in writing the TIOA file following the format specified in .jj file. It should be noted that each pattern in .jj has an associated action, which is a fragment of C code that performs such action. In our case, the actions return the tokens that will be used later on by the parser.

JavaCC also generates a parser program that handles the input source using the tokens identified by the lexical analyzer. These tokens are checked in the order in which they are listed in the lexical analyzer. 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 storing TIOA information to its corresponding data structures. For example, the data structures can be a vector list that contains the locations of TIOA, a vector list of the transitions of TIOA, etc.

The textual specification file of the TIOA in Figure 2 is as following:

```
Source: l0
Destination: l1
Action: ?ln
Constraint : x <= 1
Reset clk : x

Source: l1
Destination: l0
Action: !Out
Constraint: x == 1
Reset clk: x
```

Overall, the compiler used in our tool has the following structure:

![Diagram](image-url)

**Figure 3. Compiler Structure of the Tool.**

The parser is the main component that the second, third and fourth parts of the tool depend on.

### 2.2 The Sampler

The sampler is the first step in a series of transformations performed by our tool. The objective of this step is to sample the given TIOA specification using a suitable granularity, in order to extract a finite behavior that is easily testable. The resulting behavior is described by a finite automaton, called Grid Automaton (GA for short). In memory, a GA is represented by a vector list that contains the states of GA and the transitions of each state. Each state of GA is characterized by a location, and a value for each clock used. The granularity of sampling is the amount of time (i.e., the time delay) by which the time space of the system under test is sampled to determine the time points at which the system is controlled and observed. The sampling of the time space of the system under test is necessary because we adopt dense/continuous time model (Dill 1990) to describe and specify real-time systems. In dense time models, the events of the system are supposed to occur at any time value in the set of real numbers (i.e., the clock variables take their values in the set of real numbers). So, without sampling, the states space of the system is infinite and therefore cannot be tested. The definition of the Grid Automaton is as follows.

**Definition 2: Grid Automaton**

Let \( A = (IA, OA, LA, l0A, CA, TA) \) be a TIOA. The Grid Automaton (GA) of \( A \) is a finite input output automaton \( GA = (IGA, OGA, SGA, s0GA, TGA) \), where:

- \( IGA = IA \cup \{g\} \), where \( g \) is a special time delay (\( g \) is a rational number).
- \( OGA = OA \).
- \( SGA \) is finite set of system states. Each state is a pair \((l, v)\), where \( l \in LA \) and \( v \) is a clock valuation in which the value of each clock of \( A \) is a multiple of \( g \).
- \( s0GA \) is the initial state that consists of the initial location \( l0A \) with all clocks values set to zero.
- \( TGA \) is a finite set of Transitions. Each transition consists of a source state, an action (input, output, or time delay \( g \)), and a destination state. There are two types of transitions in GA: the delay transitions on time delay \( g \) and the explicit transitions on input and output actions. Each state has an outgoing delay transition on time delay \( g \). However, a state \((l, v)\) has an outgoing explicit transition on input or output action \( a \) if and only if there is a transition \( l \rightarrow l' \) in \( A \), and \( v \) satisfies the clock guard \( G \). After the execution of a delay transition on \( g \), the value of each clock is incremented with \( g \) time-units. However, after the execution of an explicit transition, the value of each clock in \( \lambda \) in the corresponding transition in \( A \) is set to zero.
The algorithm of sampling consists of going through all the locations and transitions of each location in TIOA and checking if the time constraints of these transitions are satisfied. If the time constraint of a transition in TIOA is satisfied, we create a transition in GA as well as its corresponding source and destination states. More specifically, the construction of GA is done as follows: First of all, the number of clocks is obtained to calculate the granularity of sampling. We pointed out in that the granularity should be \( \frac{1}{2} \) if the number of clocks is one; otherwise it should be \( \frac{1}{n+2} \), where \( n \) is the number of clocks in TIOA. Secondly, the initial grid state (i.e., the initial location of TIOA with all clocks set to zero) is constructed. Thirdly, we handle all the generated states of GA starting with the initial state. For each state, on the one hand, we generate all the reachable states by repetitive delay transitions on the granularity of sampling until the values of all clocks become irrelevant. On the other hand, we go through all the transitions from the location of the chosen state and we check if the time constraint of each of them is satisfied by the values of clocks in the state being handled. For each transition whose time constraint is satisfied, we generate a new state that corresponds to the destination state of that transition along with the transition between the chosen state and the new generated state. The handling of states stops when no state remains untreated. It should be noted that the user can choose a granularity different from the default one (i.e., \( \frac{1}{n+2} \) if \( n > 1 \)). Indeed, as we will see later on, the tester can decide to use the GCD of the time constraints bounds (or any other rational number) as a granularity of sampling in order to reduce the test cases to be generated. Of course, choosing a granularity other than the default one mentioned above may compromise the fault coverage of the tests generation method.

An example of GA for the TIOA of Figure 2 is shown in Figure 4. Here, the granularity of sampling is equal to 1 (i.e., 1 is the value of \( g \) in definition 2).

### 2.3 The Converter

The converter is responsible of transforming the GA into FSM by coupling each output with the appropriate inputs or time delays that precede it. This conversion is done in order to benefit from the existing state identification techniques that were developed to generate efficient test cases from FSM specifications. In other words, the conversion step aims to reuse/adapt existing efficient FSM based techniques for generating timed test cases and consequently achieving higher fault coverage.

The algorithm of the converter can be informally stated as follows. We go through all the transitions of the GA and look for the transitions labeled with an input or a delay. For each of these transitions, we check if it is followed by a transition on an output. If such output transition exists, we merge both transitions by coupling their input (or delay) and output, and by removing the intermediary state; otherwise, we set to NULL the output part of the transition on input or delay. If a transition on an input or a delay is followed by more than one transition on an output, all those outputs should be combined and coupled with the input of the transition. As an example, Figure 5 shows the FSM corresponding to the GA of Figure 4.

![Figure 5: A Minimized FSM for Figure 4](image)

In addition to the transformation of GA into FSM, the converter is also responsible of minimizing the resulting FSM. The minimization consists of removing the equivalent states in order to apply state identification techniques for tests generation. Two states are said equivalent if they have the same behavior in terms of inputs and outputs sequences. The minimization algorithm can be summarized as follows. We go through all pairs of states \((l, v)\) and \((l', v)\) such as \(l\) is not equal to \(l'\), and we check recursively if they accept the same input/output sequences (or traces). If one of the two accepts a trace and the other one does not, then they are judged not to be equivalent. So, they are distinguishable by at least an entry sequence. Such entries are used in computing \( Wi \) sets (see section 2.4). 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 to be equivalent. In case of equivalence, we remove one of the two states and its
children as well as all the outgoing transitions from those states, and all the incoming transitions of the removed state are oriented to the equivalent state. The equivalent states are removed to avoid having redundant information. The FSM of Figure 5 is already minimized and so does not change after running the minimization algorithm.

2.4 The Generator

The generator is responsible of the derivation of test cases from the minimized FSM that results from the application of the converter. The generator is based on a FSM state identification technique, namely: the Generalized Wp-method (G.V.Bochmann et al. 1991).

The generalized Wp-method generates test sequences from both nondeterministic and deterministic FSMs. It basically extends Wp-method to non-deterministic FSM (NFSM for short). The Generalized Wp-method consists of two phases. The first phase checks that all 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 correct output and state transfer. The test sequences of phase 1 consist of the concatenation of Q with W (i.e., Q,W), where Q is the states cover set and W is the state characterization set. However, the second phase checks the implementation for all the transitions defined by the specification which were not checked during the first phase. The test sequences of phase 2 consist of the sequences of the transitions cover set \( P \) (i.e., \( P = Q.I \), where I is the set of inputs) which are not contained in Q, concatenated with the corresponding Wi. The W set is the concatenation of the identification set Wi of each state \( S_i \) in the FSM. Figure 6 shows an FSM and the results obtained by applying the generalized Wp-method. For more details on Wp-methods, the reader is invited to consult (G.V.Bochmann et al. 1991; Susumu Fujiiwara et al. 1991).

![Figure 6. Specification S](image)

**Figure 6. Specification S**

\[ Q = \{\text{empty}, a, b\} \]
\[ P = \{\text{empty}, a, b, a.b, a.a, b.a, b.b\} \]
\[ W = \{a, b\}, W_0 = \{b\}, W_1 = \{a\}, W_2 = \{a, b\}, \]
\[ \text{Phase 1}, Q_W = \{r.a, r.b, r.a.a, r.b.a, r.b.b\}, \]
\[ \text{Phase 2}, (P-Q).\{W_i\} = \{r.a.b.a, r.a.b.b, r.a.a.a, r.a.a.b, r.b.a.a, r.b.b.a, r.b.b.b\} \]

3. Case Studies

Many real examples were applied to our tool and interesting results were obtained. The tool successfully generates test cases for many real-time systems with different sizes. The size of a real-time system (i.e., the size of its TIOA specification) is defined in terms of the number of locations, the number of transitions, the number of clocks, and the values of the bounds of time constraints. Moreover, the test cases generated by our tool have good/full fault coverage. In fact, the level of fault coverage depends on the granularity used to sample the TIOA. Indeed, if the granularity is \( \frac{n+1}{2} \) then the test cases will have full fault coverage. If the granularity is bigger than \( \frac{n+1}{2} \), then the fault coverage of the test cases will be partial but good enough. In what follows, we describe four examples of RTS and we report the results obtained by the application of our tool. The tool was run with different granularities: \( \frac{n+1}{2} \), 1, the greatest common factor of the bounds of the time constraints, or any rational number. The results obtained by running our tool on each of the examples are presented in terms of the number of states in GA, the number of test cases, and the response time.

3.1 Telephone System

The telephone system (D. Clarke & Lee 1997) is responsible of establishing the connection between two parties after receiving the digits of the phone number dialed by the caller. Figure 7 shows a TIOA for such system. It has five locations and the initial location is “Idle”. When the user picks up the phone, the telephone system outputs the dial tone and initializes the clock to zero, and enters the “Dial Tone” location. 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 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” location is reached and the connection is established. One clock is used to evaluate the signal, dialing time, and connecting time. The results of the telephone system are shown in Table 1. For the generator to derive test cases from the FSM produced by the converter, it should implement the algorithms to find the sets P, Q, W, and Wi. To find the Q set, the generator implements the Dijkstra’s algorithm (Ravindra K. Ahuja & al., 1993; Waldura 2002) to determine the shortest path from the initial state of the FSM to each other state. To this end, each transition in the minimized FSM is assigned a cost equal to 1. After the determination of the Q set, the P set can be simply obtained by concatenating Q with the input set of the FSM. For the Wi sets, they can be recursively determined during the minimization process. When two states are found to be not equivalent, the distinguishing sequence is added to the Wi set of each of them. Once all Wi are determined, the W set is simply the concatenation of all of them. After all these sets have been found by the generator, the test
cases are obtained as above.

**Figure 7. Telephone System (1)**

<table>
<thead>
<tr>
<th>Granularity</th>
<th>GA States</th>
<th>Minimized FSM States</th>
<th>Test Cases Generated</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>½</td>
<td>160</td>
<td>97</td>
<td>344</td>
<td>Real: 7m7523s</td>
</tr>
<tr>
<td>1</td>
<td>85</td>
<td>51</td>
<td>183</td>
<td>Real: 0m9.90s</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>14</td>
<td>53</td>
<td>Real: 0.844s</td>
</tr>
</tbody>
</table>

**Table 1. Telephone System Results(1)**

The same telephone system may have other dialing-time requirements. For example, the time between each successive pair of digits can be a 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 20 seconds. Figure 8 shows the behavior of the system in terms of 14 locations with their transitions. The results obtained from the tool for this system are summarized in Table 2.

**Figure 8. Telephone System (2)**

<table>
<thead>
<tr>
<th>Location</th>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>?image</td>
<td>x=0</td>
</tr>
<tr>
<td>S2</td>
<td>?sound</td>
<td>x=0, y=0</td>
</tr>
<tr>
<td>S3</td>
<td>!ackAll</td>
<td>0&lt;=y&lt;=5</td>
</tr>
<tr>
<td>S4</td>
<td>!reset</td>
<td>x=0</td>
</tr>
</tbody>
</table>

**Table 2. Telephone System Results(2)**

3.2 Small Multimedia System

The TIOA presented in Figure 9 shows the behavior of small Multimedia System (H. Fouchal et al., 1998). When the system is in location “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 location “S2” waiting for the sound as a second input at most 2 seconds after receiving the 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. After sending the acknowledgment, the clocks are again reset and the system goes back to its initial location. Table 3 shows the results corresponding to this system.

**Figure 9. Multimedia System**

<table>
<thead>
<tr>
<th>Granularity</th>
<th>GA States</th>
<th>Minimized FSM States</th>
<th>Test Cases Generated</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>¼</td>
<td>88</td>
<td>74</td>
<td>249</td>
<td>Real: 3m0317s</td>
</tr>
<tr>
<td>1</td>
<td>28</td>
<td>22</td>
<td>85</td>
<td>Real: 0m4.10s</td>
</tr>
</tbody>
</table>

**Table 3. Multimedia System Results**

3.3 Media Synchronization Protocol

The media synchronization protocol (T. Higashino et al. 1999) is a protocol, which is responsible of synchronizing real time continuous media such as video stream when the transfer rate quickly changes. Figure 10 shows the behavior of the protocol. The locations: “S0”, “S1”, “S4” and “S5” try to send data continuously at the fixed rate to the locations: “S2” and “S6”. Here, the symbols a, b, d, x and y are constant parameters specified by the designer, where:

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

Figure 10. Media Synchronization Protocol

First, “S0” sends to “S2” a video packet with a timestamp representing its sending time. The receiving location “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 duration necessary to finish playing [a, b]. After that, the receiving location “S6” carries out the above work continuously. Moreover, it synchronizes the playing rate with the sending location. 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 the 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. Again, the TIOA description in Figure 10 is deterministic, which has 5 clocks.

To execute the media synchronization protocol example and have the test suite, the constant parameters must be specified with some real values. We chose the values of the parameters of as shown in Table 4. The first two rows of the table assumes 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. However, the second two rows of the table indicates that 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.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>States GA</th>
<th>Minimized FSM States</th>
<th>Test Cases Generated</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3082</td>
<td>1967</td>
<td>5000</td>
<td>Real:320m</td>
</tr>
<tr>
<td>5</td>
<td>232</td>
<td>153</td>
<td>435</td>
<td>Real:9.062s</td>
</tr>
<tr>
<td>1</td>
<td>11, 795</td>
<td>9340</td>
<td>More than one day and there is no test cases.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>649</td>
<td>499</td>
<td>1765</td>
<td>Real:2m8143s</td>
</tr>
</tbody>
</table>

Table 4. Media synchronization protocol results

3.4 Railroad Crossing System

In the railroad system (Cardell-Oliver 1999), 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).

The railroad crossing system is composed of three deterministic TIOA descriptions, which have shared events between them,, and the timing assumptions of the system are as follows:

- The train (Figure 11) enters the crossing within an interval of [60, 120] seconds after sending a (? Near) message to the 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 11. Train

- The gate (Figure 12) closes within 60 seconds of receiving the instruction (?Lower) from the controller. A gate then triggers the opening event (Up) within an interval of 0 to 60 seconds of receiving the instruction (?Raise) from a controller.
The controller (Figure 13) instructs 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.

![Gate Diagram](image1)

**Figure 12. Gate**

- The controller (Figure 13) instructs 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.

![Controller Diagram](image2)

**Figure 13. Controller**

The results obtained by applying the tool on the controller are shown in Table 5.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>GA States</th>
<th>Minimized FSM States</th>
<th>Test Cases Generated</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>½</td>
<td>968</td>
<td>966</td>
<td>4106</td>
<td>Real: 1h4m045s</td>
</tr>
<tr>
<td>1</td>
<td>488</td>
<td>486</td>
<td>2064</td>
<td>Real: 8m48s</td>
</tr>
<tr>
<td>2</td>
<td>248</td>
<td>246</td>
<td>1044</td>
<td>Real: 1m125s</td>
</tr>
<tr>
<td>5</td>
<td>104</td>
<td>102</td>
<td>432</td>
<td>Real: 3.047s</td>
</tr>
<tr>
<td>60</td>
<td>16</td>
<td>14</td>
<td>58</td>
<td>Real: 0.828s</td>
</tr>
</tbody>
</table>

**Table 5. Controller’s Results**

### 3.5 Results Analysis

In the tables showing the results, we have 5 columns that describe the following characteristics: the chosen granularity for each system, the number of grid states and the number of states in the minimized FSM, the number of test cases, and the required time to obtain the results. In each system row, we have the granularity used to sample the TIOA of the system being tested. The machine that we have run the tool on is Intel P4 2.0GHz, 512 MB RAM, Windows XP home edition.

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

  - If the tool is applied with real-time systems of three clocks to five clocks, which is the rare reality 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 RTS with large number of clocks and integer constants. The latest case is not important because almost all real-time systems can be described by at most two clocks and acceptable constraints bounds.

  - For any input transition, there exists at least one test case that covers the time space of the transition when the granularity is $\frac{1}{n+2}$. If we use bigger granularities, the number of test cases will be significantly reduced but the fault coverage of the method might be compromised (i.e., some faults may not be detected by the test cases generated). So, the tester has to make a trade-off between the number of test cases to be generated and the level of fault coverage to be reached.

  - The most time execution is spent on generating the Q set and the test suite.

  - It is to be noted that when the granularity of 1 is used instead of $\frac{1}{n+2}$, then the test cases are approximately reduced by $(1-\left(\frac{1}{n+2}\right))$. 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 clock x for many transitions. This leads to new states with different locations of x equal to zero and different values of clock y.

### 4. Conclusion & Future Work

Real-time systems are more challenging to test than non-real-time systems. We presented a tool to generate test cases for RTS based on Timed Wp-method as well as some case studies. Our tool is scalable and efficient enough to produce the required test cases within acceptable response time. In order to generate the test cases, the tool is divided into four parts: the parser, the sampler, the converter and the generator. The tool optimizes the test cases by removing the test cases that are prefixes of others (e.g., the test case “a.b.a” is a prefix of the test case “a.b.a.a” and so should not be included in the final set of test cases).

We are currently working on the formalization of the relationship between the sampling granularity, the fault model, and the fault coverage. We are also planning to extend our methodology to deal with specifications that contain data variables and constraints on these data variables.

### 5. References

Transactions on Software Engineering (TSE), 28(11), 1023-1038.


