Test Generation for Time Critical Systems: Tool and Case Study

Brian Nielsen and Arne Skou

Aalborg University
Department of Computer Science
Fredrik Bajersvej 7E
DK-9220 Aalborg, Denmark
Email: {bnielsen | ask}@cs.auc.dk

Abstract

Generating timed test sequences by hand is error-prone and time consuming, and it is easy to overlook important scenarios. The paper presents a tool based on formal methods that automatically computes a test suite for conformance testing of time critical systems. The generated tests are selected on the basis of a coverage criterion of the specification. The tool guarantees production of sound test cases only, and is able to produce a complete covering test suite. We demonstrate the tool by generating test cases for the Philips Audio Protocol.

1 Introduction

Testing consists of executing a program or a physical system with the intention of finding undiscovered errors. *Conformance testing* is a black box approach that aims at checking that the behavior of an implementation complies to the behavior of the specification. Testing is the most dominating validation activity used by industry today. However, there are two well documented problems with the current state of the art. First, testing is very expensive; depending on application as much as 20% to 50% of the total development time is spent on testing. Second, despite this effort and the hard work of dedicated test engineers, a significant amount of errors remain.

A potential improvement that is being examined by researchers is to provide theoretically well founded tools that automate test generation and execution. Test generation tools are usually much faster than humans, and has the potential for being more systematic than humans, and thereby generate important tests that could easily be overlooked.

This approach has experienced some level of success [5, 15, 19, 21], and commercial test generations tools are emerging [5, 19]. However, these tools do not address real-time systems, or only provide a limited support for testing the timing aspects. They often abstract away the actual time at which events are supplied or expected, or do not select these time instances thoroughly and systematically. Exhaustive testing is usually infeasible, and because a real time system consists of an enormous amount of time instances that could be relevant to test, it is not likely that an arbitrary or random choice of such time instances will result in good coverage. It is therefore important to make good decisions of *when* to deliver an input to the system, and *when* to expect an output.

This paper presents a *tool* for automatic generation of timed tests from a restricted class of dense, but possibly non-deterministic, timed automata specifications. The technique is applied to a realistic case: a Phillips Audio Protocol specification.

The test cases are generated *systematically* from a coverage criterion of the specification. The state space of the specification is partitioned into coarse grained equivalence classes which preserve essential timing and synchronization information, and a few tests for each class are generated. This approach is inspired by sequential black-box testing techniques frequently referred to as domain- or partition testing [3]. We regard the clocks of a timed specification as (oddly behaving) input parameters. Our technique *guarantees* that *every* reachable equivalence class will be covered by a set of relevant tests, and that every generated test is *sound*, i.e., failure to pass the test implies that the system under test is non-conforming.

Analyzing a timed specification is no easy task, and nearly impossibly to do by hand, even for moderate size specification. We therefore employ efficient automatic symbolic reachability techniques based on constraint solving that have recently been developed for model checking of timed automata [9, 13].

The emphasis of this paper is our test generation tool

and an application thereof. The underlying algorithms and testing theory, which is based on Hennessy's work [16], are described in detail in [18, 17]. Section 2 summarizes the related work. Section 3 presents the specification language, and Section 4 gives an overview of the techniques implemented in our tool, which is further described in Section 5. Section 6 provides a first, but realistic case study. Section 7 concludes the paper and suggests future work.

2 Related Work

Springintveld et al. proved in [22] that *exhaustive* testing wrt. *trace equivalence* of *deterministic* timed automata with a *dense time* interpretation is theoretically possible, but highly infeasible in practice. Another result generating checking sequences for a discretized *deterministic* timed automaton is presented by En-Nouaary et al. in [10]. Although the required discretization step size $(1/(|X|+2)$, where $|X|$ is the number of clocks) in [10] is more reasonable than [22], it still appears to be too small for most practical applications because too many tests are generated.

Clarke and Lee [8]—like we—propose domain based test selection for real-time systems. Although their primary goal of using testing as a means of approximating verification to reduce the state explosion problem is different from ours, their generated tests could potentially be applied to physical systems as well. Compared to timed automata their language for specification of time requirements appear very restricted.

Castanet et al. presents in [7] an approach where timed test *traces* can be generated from timed automata specifications. Test selection must be done *manually* through engineer specified test purposes (one for each test) themselves given as deterministic acyclic timed automata. Such explicit test selection reduces the state explosion problem during test generation, but leaves a significant burden on the engineer. Our goal has been fully automatic test generation.

Cardell-Oliver and Glover showed in [6] how to derive checking sequence from a *discrete time*, *deterministic*, timed transition system model. No test selection is performed in the time domain. Finally, test generation from a discrete time temporal logic is investigated by [15].

3 Event Recording Automata

Two of the surprising undecidability results from the theoretical work on timed languages described by timed automata are that 1) a non-deterministic timed automaton cannot in general be converted into a deterministic (trace) equivalent timed automaton, and 2) trace (language) inclusion between two non-deterministic timed automata is undecidable [2]. Thus, unlike the untimed case, deterministic and non-deterministic timed automata are not equally expressive. The Event Recording Automata model (ERA) was proposed by Alur, Fix, and Henzinger in [2] as a determinizable subclass of timed automata, which enjoys both properties. This property is highly desirable for systematic generation of Hennessy based tests and their assigned verdicts.

Definition 1 *Event Recording Automaton*:

1. An ERA $\mathcal{M}$ is a tuple $\langle Act, N, l_0, E\rangle$ where Act is the set of actions, N is a (finite) set of locations, $l_0 \in N$ is the initial location, and $E \subseteq N \times G(X) \times Act \times N$ is the set of edges. The term *location* denotes a node in the automaton, and the term *state* denotes the semantic state of the automaton also including clock values.

2. $X = \{x_a \mid a \in Act\}$ is the set of clocks. The guards $G(X)$ are generated by the syntax $g ::= \gamma \mid g \wedge g$ where γ is a constraint of the form $x_1 \sim c$ or $x_1 - x_2 \sim c$ with $\sim \in \{\leq, <, =, >, \geq\}$, c a non-negative integer constant, and $x_1, x_2 \in X$.

Like a timed automaton, an ERA has a set of clocks which can be used in guards on actions, and which can be reset when an action is taken. In ERAs however, each action a is uniquely associated with a clock x_a, called the *event clock* of a. Whenever an action a is executed, the event clock x_a is automatically reset. No further clock assignments are permitted. The event clock x_a thus *records* the amount of time passed since the last occurrence of a. In addition, no internal τ actions are permitted. These restrictions are sufficient to ensure determinizability [2]. Examples are given in Figure 4 and Figure 8.

4 Test Generation Technique

4.1 Partitioning

Since exhaustive testing is generally infeasible, it is important to systematically select and generate a limited amount of tests. A test *selection criterion* (or coverage criterion) is a rule describing what behavior or requirements should be tested. *Coverage* is a metric of completeness with respect to a test selection criterion. Our *stable edge set criterion* partitions the state space of the specification into coarse equivalence classes, and requires that the test suite for each class makes a set of required observations of the implementation when it is expected to be in a state in that class.

The states (a pair consisting of automaton locations and clock valuations) of the automaton are partitioned such that two clock valuations belong to the same equivalence class iff they enable precisely the same edges from the set of

states that the automaton currently possibly can occupy, i.e. the states are equivalent wrt. the enabled edges. We justify this partitioning by the following observations:

- Because the enabled edges change when the specification moves from one equivalence class to another by executing an action or letting time pass, the implementation must somehow perform a matching action or an internal timeout to change the enabled edges, and it must consequently be checked that the implementation responds correctly after that internal action or timeout.
- Because the enabled edges are the same we believe that it is reasonable to expect that the implementation treats these states *uniformly*. Interior and extreme clock values for the equivalence class can be used to support this hypothesis.
- The partitioning has the nice formal property that all states in the same equivalence class also satisfy the same basic Hennessy observations.
- This partitioning is based on the guards that actually occur in a specification, and is therefore much coarser than e.g., the region partitioning which is based on the guards that could possibly occur in an automaton according to the syntax in Definition 1.

In conclusion, it is more important to test different classes than it is to test the same one many times.

4.2 Symbolic Analysis

Densely timed automata cannot be analyzed by enumerative finite state techniques, but must rather be analyzed symbolically [1]. We employ the so-called *zone* and *difference bound matrix* techniques [9] that have proven efficient [13] for model checking of timed automata.

A zone z over clocks $x \in X$ is a constraint system consisting of conjunctions of linear inequalities on clock variables of the form :

$\{x_i - x_j \prec c_{ij} \mid i,j \leq n\} \cup \{a_i \prec x_i\} \cup \{x_i \prec b_i\}$, where $\prec \in \{<, \leq\}$, c_{ij}, a_i, b_i are integers including ∞, and $x_i, x_j \in X$.

Zones can be represented efficiently by the *difference bound matrix* (DBM) data structure [9]. A DBM represents clock difference constraints of the form $x_i - x_j \leq c_{ij}$ by a $(n+1) \times (n+1)$ matrix such that c_{ij} equals matrix element (i,j), and where n is the number of clocks. To represent constraints of the form $x_i \leq c$, DBMs use a special zero clock $\mathbf{0}$ which has the constant value 0. Figure 2 shows an example of a DBM.

A concrete state of a timed automata can be represented by a pair $(\bar{l}, \bar{u})$ consisting of location (vector) $\bar{l}$ and clock valuation $\bar{u}$ st. $\bar{u}(x)$ is the current value of clock x. A symbolic state $[\bar{l}, z]$ is a pair consisting of a location (vector) and zone z. It represents the possibly infinite set of states $\{(\bar{l}', \bar{u}) \mid \bar{l}' = \bar{l} \wedge \bar{u} \in z\}$. An efficient set of operations on zones allows the following to be computed:

	$\mathbf{0}$	x_1	x_2
$\mathbf{0}$	$\times$	-5	-1
x_1	7	$\times$	4
x_2	3	-2	$\times$

Figure 2. DBM representation of the constraint $z = x_1 - x_2 \leq 4 \ \wedge \ x_2 \leq 3 \ \wedge \ x_1 \geq 5$

- The symbolic state that results by taking an edge from a given source symbolic state can be computed. Such a forward execution sequence is started in the initial state of the automaton $(\bar{l}_0, \bar{0})$.
- The reachable state space can be computed by checking, at each forward step as above, whether the target symbolic state is included in one previously visited. If so, it can be concluded that no new states can be reached from it, and hence further exploration is not needed.
- Given a symbolic path to a symbolic state, a concrete trace leading to it can be computed.

To ensure soundness of the produced tests, symbolic (reachability) analysis is needed to select only states for testing that are reachable, and to compute only timed traces actually in the specification.

4.3 Timed Trace Computation

When a desired target symbolic state is reached, it can be concluded that all concrete states in the symbolic target states are reachable. However, it is not ensured that *all* states along the path of symbolic states used to reach it, necessarily will end in a state in the target symbolic state, but only that *some* of the states traversed underway will end up in the target state. Therefore, when a trace leading to the desired target is to be computed, the trace must pass through the states only that can reach the desired target. It is relatively straight forward to compute the preconditions for the required subsets by back-propagating the zone constraints of the target states back along the path used to reach it. Back propagation results in a strengthened symbolic trace representing a possibly infinite set of concrete traces leading to the target.

From this set the tester can choose a specific trace by controlling when actions are offered and observed, i.e., by

choosing the specific delay to wait between actions. This process is started at the initial state. The possible delays which can be chosen are defined by the strengthened symbolic states. Let D be the set of possible delays before an action. There are three immediate strategies for choosing delays:

1. Choose the smallest delay $d \in D$. This checks the *promptness* of the implementation by executing the succeeding action at the earliest time possible in the current trajectory.

2. Choose the delay (possibly stochastically) to be in the middle of D. This checks the *persistence* of the implementation, i.e., that the succeeding action can be executed in the interior of its enabling interval.

3. Choose the delay to be the largest delay in D. This tests the *patience* of the system, i.e., that the succeeding action is also executable at the latest required enabled time.

Of the above strategies, it seems most important to check the promptness of the system as this checks for missed deadline errors, which are common in real-time systems. But also the patience may be important, since this may detect errors where a timer times out prematurely.

4.4 Overall Algorithm

The test generation procedure, outlined in Algorithm 3, first constructs the equivalence classes and stores them in a data structure which we refer to as the *equivalence class graph*. It preserves all timed traces of the specification, and furthermore preserves the required synchronization information for our timed Hennessy tests. All timed Hennessy tests that the specification passes can thus be generated from this graph.

Algorithm 3 *Overall Test Case Generation Algorithm*:
input: ERA specification $\mathcal{S}$.
output: A complete covering set of timed Hennessy tests.

1. Compute $\mathcal{S}_p =$ Equivalence Class Graph$(\mathcal{S})$.
2. Compute $\mathcal{S}_r =$ Reachability$(\mathcal{S}_p)$.
3. Label every symbolic State $S \in \mathcal{S}_r$ with the Hennessy observations satisfied by S.
4. Traverse $\mathcal{S}_r$. For each reached equivalence class S in $\mathcal{S}_r$:
   (a) Choose a state to be tested $s \in S$
   (b) Compute a timed trace σ from initial state to s.
   (c) Make test cases to be passed: For each Hennessy observation o in S, trace σ followed by observation o is a sound test.

5 Tool Facilities

We have implemented our approach and algorithms in a prototype tool called RTCAT. RTCAT inputs an ERA specification in AUTOGRAPH format [20]. A specification may consist of several ERAs operating in parallel, and communicating via shared clocks and integer variables, but no internal synchronization is allowed as stated in Section 3. Other features include:

Termination: By default, the entire equivalence class graph is constructed. Reachability graph construction terminates when no further equivalence classes can be reached. The result is generation of a complete covering test suite.

We have also implemented a few pragmatic strategies for handling specifications whose reachability or equivalence class graphs are too large to be completely computed, stored, or tested. Construction of both graphs can also be terminated by specifying a maximum *trace depth*, using *bit-state hashing*, or both. Bit-state hashing [11] is a technique that limits the number of nodes in a graph, and is believed to result in a better (under) approximation of the state space than random exploration, which has a tendency of confining itself to small parts of the state space.

Construction Order: Both breadth first and depth first construction of the equivalence class and reachability graphs are implemented. The tests for a given equivalence class are generated the first time it is reached during forward reachability analysis. Consequently, the traversal order may affect the number and length of tests generated.

Our experience suggests that breadth first construction results in the most economic test suite in terms of length. Depth first results in slightly fewer test cases but much longer test suites, and should be used when a covering test suite should be obtained that also tests the behavior after relatively long sequences of actions.

Test Structure: Tests can be constructed either as individual timed Hennessy tests (Algorithm 3) or as test trees which merge the individual tests when possible.

Trace Generation: Timed traces can be generated using prompt, interior, or patience selection as described in Section 4.3.

Extreme value selection is currently not supported, but can easily be implemented. The prototype operates in four distinct phases, i.e., the preceding must be completed before a new is started: parsing and initialization, equivalence class

graph construction, reachability graph construction, and finally timed trace computation and output of the test suite to a file in DOT format [12]. RTCAT occupies about 22K lines of C++ code, and is based on code from a simulator for timed automata (part of an old version of the UPPAAL toolkit [14]). Its AUTOGRAPH file format parser was reused with some minor modifications to accommodate the ERA syntax. Also its DBM implementation was reused with some added operations for zone extrapolation and clock scaling.

6 A case study

6.1 Example 1

The ERA example in Figure 4a demonstrates that computing test cases from a timed automata specification by hand is non-trivial, even for very small specifications. For example, to compute a test that visits the edge $s_1 \xrightarrow{a?, X_a \leq 1} s_0$, the edge $s_0 \xrightarrow{a?, 1 < X_a < 2} s_0$ must be visited at least three times in succession for the guard on the b edge to become satisfiable, Furthermore, the b edge must be visited before X_a equals 1 time unit; otherwise, the guard on the succeeding $s_1 \xrightarrow{a?, X_a \leq 1} s_0$ edge is not satisfiable. The tool generates the test automaton shown in Figure 4b; its locations are labeled with the visited location of the specification, and the test verdict (**p**=pass, **f**=fail) to be given if the test execution stops in that location. A total of 12 such tests is generated to cover the specification. It is thus easy to overlook an important scenario.

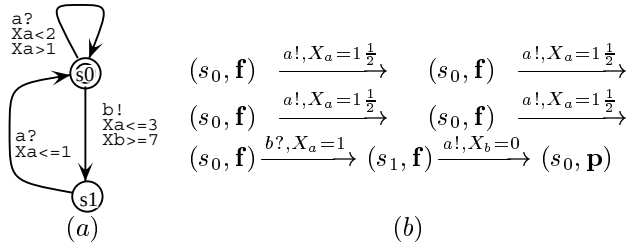

Figure 4. Simple ERA example

6.2 Example 2: Philips Audio Protocol

The Philips Audio Protocol is a dedicated protocol for exchanging control information between audio/visual consumer electronic units. Consequently, the protocol must be simple and cheap to implement. The data is Manchester encoded, and transmitted on a shared bus implemented as a single wire. There are two interesting aspects of this protocol. One is that a certain tolerance is permitted on the timing of events to compensate for drift of hardware clocks and CPU contention. Philips permits a $\pm 5\%$ tolerance on all the timing, while still being able to decode the transmitted signal correctly. The second aspect is that the collisions of messages on the bus must be detected. The protocol was first studied by Bosscher et al. in [4]. It was here proven formally that the signals can be correctly decoded if tolerances are less than $\frac{1}{17}$. The protocol has since been studied numerous times in the context of model checking.

The goal of generating tests for the protocol is to compute a test suite that can be used to determine if a given audio component implements the Manchester encoding and collision detection correctly, and within the allowed tolerances.

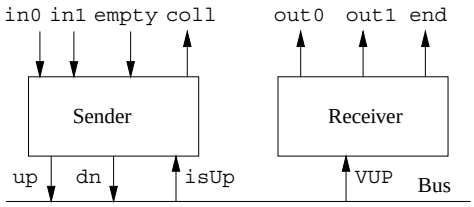

Figure 5. Overview of the Philips audio protocol.

A station is equipped with a module for encoding and transmitting data on the bus, and a module for receiving and decoding the data. An overview of the protocol is shown in Figure 5. The sender obtains the bit stream to be transmitted via three actions: `in0`, `in1`, and `empty`, respectively representing a zero-bit, a one-bit, and an end of message delimiter. The sender Manchester encodes these bits, and uses the actions `up` and `dn` to drive the bus voltage high and low respectively.

The bus works as a logical or, so whenever a station drives the bus high, the bus will be high even if other stations previously has set it low. A sender can detect collision by checking that the bus is indeed low when it is itself sending a low. The `isUp` action is used for this purpose. If a collision is detected, the upper protocol layer is informed via the `coll` action.

The receiver informs the upper layer of the decoded bits via the `out1`, `out0`, and `end` actions. Philips uses rising edge triggering to decode the electrical signal. A rising edge is indicated to the receiver by the `VUP` action. To decode the signal using only rising edge triggering as required by Philips, messages must start with a logical one, and be odd in length.

Using Manchester encoding, illustrated in Figure 6, the time axis is divided into equal sized *bit slots*. In every bit slot one bit can be sent. A bit slot is further halved into two intervals. A logical zero is represented by a low voltage on

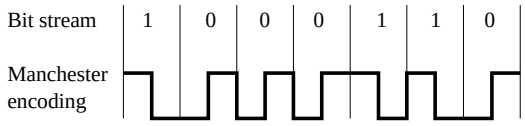

Figure 6. Manchester encoding of the bit stream 1000110.

the wire during the first interval of a bit slot, a rising edge at half the bit slot, and high voltage during the last interval. A logical one is represented by a high during the first interval, followed by falling edge, and a low through the last half.

A bit slot in the Philips protocol is $888\mu s$ long. In the modeling we use quarters of bit slots, denoted q, equaling $222\mu s$. The basic constants used in the model, and the derived tolerance levels are summarized in Table 7.

Symbol	Value	Meaning			
q	2220	one quarter of a bit slot ($220\mu s$)			
d	200	Detection 'just' before up ($20\mu s$)			
g	220	'Around' 25% and 75% of the bit-slot ($22\mu s$)			
w	80000	Station Silence ($8ms$)			
t	0.05	Tolerance (5%)			
A1min	2000	q-g	A1max	2440	q+g
A2min	6440	3q-g	A2max	6880	3q+g
Q2	4440	2q	Q2minD	4018	2q(1-t)-d
Q2min	4218	2q(1-t)	Q2max	4662	2q(1+t)
Q3min	6327	3q(1-t)	Q3max	6993	3q(1+t)
Q4	8880	4q	Q4minD	8236	4q(1-t)-d
Q4min	8436	4q(1-t)	Q4max	9324	4q(t+t)
Q5min	10545	5q(1-t)	Q5max	11655	5q(1+t)
Q7min	14763	7q(1-t)	Q7max	16317	7q(1+t)
Q9min	18981	9q(1-t)	Q9max	20979	9q(1+t)

Table 7. Constants used in the ERA specification of the Philips audio protocol.

The basic operating principle of the sender, shown in Figure 8, is that it inputs a new bit while encoding the current bit, i.e., it has read a bit ahead. The important states are labeled `SXtoY`, where X represents the bit currently being generated, and Y the bit to be generated next. Observe that whenever X and Y differ, the sender waits twice the normal duration before changing the status of the wire. The ERA for the receiver can be found in [17].

To detect collisions the bus must according to Phillips be sampled 'around' three specific time points, namely after a quarter of a bit slot after starting a low signal, again after three quarters (if still transmitting a low as in the one-to-zero transition), and 'just' before setting the bus high.

The generated tests are exemplified in Figure 9. Test case 1 produces the bit string '1001', and checks whether the implementation can produce this sequence, and whether it like the specification refuses all actions at the state and time entered thereafter (s_4). If one of the offered actions are accepted, the test execution will terminate in a state s_x with **fail** verdict. Test case 2 checks whether collision detection is performed after, in this case, transmission of the single bit message '1'.

Using breadth first traversal RTCAT generates a test suite consisting of 68 test cases with a combined length of 393 steps. Using depth first traversal it generates 67 test cases with a combined length of 487 steps. The timed traces are generated using prompt selection. Generating these tests and writing them to a file took less that 2 seconds, and required less that 5 Mb memory in total. The resulting test suite is so small that it can easily be executed, and there is plenty room for generating longer test suites, and further extreme values.

The case study illustrates that test cases can be generated from a real-life case, but it has also revealed a point where our current approach can be improved. For example, in our modeling of collision detection, the sender is required to be able to synchronize with the `isUp` action at all instances in the $\pm g$ interval. This is probably not what the Philips engineers have in mind. Rather, they intend to sample the bus at some point in this interval. However, this form of *timing uncertainty* cannot be readily modeled in the current ERA language. It is possible to change the specification (by using a non-deterministic choice) such that the proper verdict (inconclusive) is assigned to the tests, but executing them will most likely result in large number of inconclusive verdicts, because the action could not be observed at the chosen time.

Also it should be noted that the timing tolerances are modeled by permitting the upper protocol layer to deliver the next bit to be transmitted at some point in the "window of opportunity". The sender is therefore required to accept bits at any time within the tolerance interval. If the interface of the actual Philips components is different from this, the test cases will not be directly executable as is. An important lesson learned is that the specification model used for test generation must accurately reflect the behavior at the interface of the component to be tested.

We conclude that our technique is applicable "as is" for strictly timed embedded controllers that are deterministic with respect to time, but that it will be important to add support for timing uncertainty.

7 Conclusions and Future Work

We have presented a prototype tool for automatic and systematic generation of test cases, and have demonstrated its applicability for generating tests via a real-life example.

The number of tests generated, and the size of systems that can be handled, suggests that the basic technique is sound and practically relevant, but we also have identified a

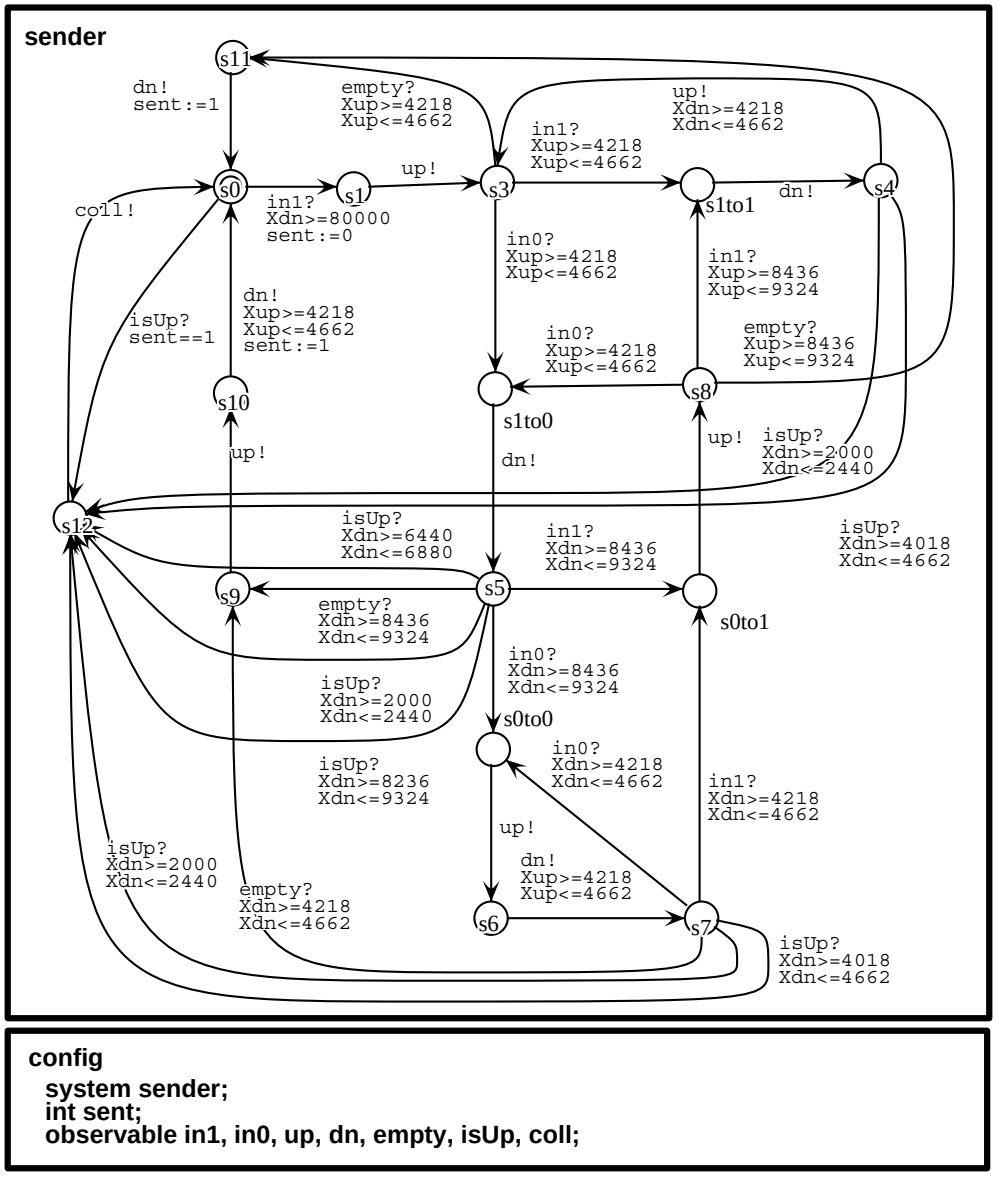

Figure 8. The sender ERA with collision detection.

Test Case 1.

$$(s_0,\mathbf{f}) \xrightarrow{in_1!,X_{up}=80000} (s_1,\mathbf{f}) \xrightarrow{up?,X_{in_1}=0} (s_3,\mathbf{f}) \xrightarrow{in_0!,X_{up}=4218} (s_{1to0},\mathbf{f}) \xrightarrow{dn?,X_{in_0}=0} (s_5,\mathbf{f}) \xrightarrow{in_0!,X_{dn}=8436}$$
$$(s_{0to0},\mathbf{f}) \xrightarrow{up?,X_{in_0}=0} (s_6,\mathbf{f}) \xrightarrow{dn?,X_{up}=4218} (s_7,\mathbf{f}) \xrightarrow{in_1!,X_{dn}=4218} (s_{0to1},\mathbf{f}) \xrightarrow{up?,X_{in_1}=0} (s_8,\mathbf{f}) \xrightarrow{in_1!,X_{up}=8436}$$
$$(s_{1to1},\mathbf{f}) \xrightarrow{dn?,X_{in_1}=0} (s_4,\mathbf{p}) \xrightarrow{Act,X_{dn}=0} (s_x,\mathbf{f})$$

Test Case 2.

$$(s_0,\mathbf{f}) \xrightarrow{in_1!,X_{up}=80000} (s_1,\mathbf{f}) \xrightarrow{up?,X_{in_1}=0} (s_3,\mathbf{f}) \xrightarrow{empty!,X_{up}=4218} (s_1,\mathbf{f}) \xrightarrow{dn?,X_{empty}=0} (s_0,\mathbf{f}) \xrightarrow{isUp!,X_{dn}=0}$$
$$(s_{12},\mathbf{f}) \xrightarrow{coll?,X_{isUp}=0} (s_0,\mathbf{f}) \xrightarrow{isUp!,X_{coll}=80000} (s_{12},\mathbf{p})$$

Figure 9. Examples of tests generated for the Philips Audio Protocol sender.

number of areas which can enlarge the application domain of our technique. It will be important to handle systems with timing uncertainty more effectively than presently done. Timing uncertainty means that an action may occur some time in an interval, but which instant is non-deterministic. Effective support will affect our technique in two areas. First, the testing theory and algorithms need to distinguish between actions that are inputs controlled by the tester or environment, and actions that are outputs controlled by the system self. Our modeling effort suggests that timing uncertainty is typically associated with outputs. Second, because the time instances of actions with timing uncertainty will not be known until runtime, and because this time affects when the next action in the test case is to be offered/observed, test cases will need to be symbolic. The timed trace will thus be instantiated at test executed time rather than as presently done at test generation time. Fortunately, both aspects appear only to require a moderate effort to incorporate, e.g., the time constraints needed to distinguish when to produce pass, fail and inconclusive verdicts is available from the computed symbolic path.

A second aspect is to generate and select test cases through manually stated test purposes, and to generate very long test cases using a guided random simulation where the probability of choosing a given delay between a pair of actions is guided by the equivalence class partitioning.

Finally, our work has focused on generating test cases. It would be very interesting to also execute them against real implementations. This will provide valuable insight in what will be a good communication model between tester and implementation in practice.

References

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 25 Apr. 1994.

[2] R. Alur, L. Fix, and T. A. Henzinger. Event-Clock Automata: A Determinizable Class of Timed Automata. In *6th Conference on Computer Aided Verification*, 1994. Also in LNCS 818.

[3] B. Beizer. *Software Testing Techniques*. International Thompson Computer Press, 1990. 2nd edition, ISBN 1850328803.

[4] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio Protocol. TR CS-R9445, CWI, Amsterdam, The Netherlands, 1994. Also in LNCS 863, 1994.

[5] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.

[6] R. Cardell-Oliver and T. Glover. A Practical and Complete Algorithm for Testing Real-Time Systems. In *5th international Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, pages 251–261, September 14–18 1998. Also in LNCS 1486.

[7] R. Castanet, O. Koné, and P. Laurençot. On the fly test generation for real-time protocols. In *International Conference in Computer Communications and Networks*, Lafayette, Lousiana, USA, October 12-15 1998. IEEE Computer Society Press.

[8] D. Clarke and I. Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[9] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, Grenoble, France, June 1989. LNCS 407.

[10] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Test Cases Generation Based on State Characterization Technique. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, December 2–4 1998.

[11] G. J. Holzmann. *Design and Validation of Computer Protocols*, chapter 11. Prentice Hall, New Jersey, U.S.A, 1991. ISBN 0-13-539925-4.

[12] E. Koustsofios and S. C. North. Drawing Graphs with dot. Technical Report `http://www.research.att.com/sw/tools/graphviz/dotguide.ps.gz`, AT&T Bell Laboratories, Murray Hill, NJ, U.S.A.

[13] K. G. Larsen, F. Larsson, P. Petterson, and W. Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *18th IEEE Real-Time Systems Symposium*, pages 14–24, 1997.

[14] K. G. Larsen, P. Pettersson, and W. Yi. UppAal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

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

[16] R. D. Nicola and M. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[17] B. Nielsen. *Specification and Test of Real-Time Systems*. PhD thesis, Department of Computer Science, Aalborg University, Denmark, april 2000.

[18] B. Nielsen and A. Skou. Automated Test Generation Timed Automata. In T. Margaria and W. Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy, April 2001.

[19] J. Peleska and B. Buth. Formal Methods for the International Space Station ISS. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, pages 363–389, 1999. Springer LNCS 1710.

[20] A. Ressouche, R. de Simone, A. Bouali, and V. Roy. The FCTOOLS User Manual. Technical Report `ftp://ftp-sop.inria.fr/meije/verif/fc2.userman.ps`, INRIA Sophia Antipolis.

[21] H. Schlingloff, O. Meyer, and T. Hülsing. Correctness Analysis of an Embedded Controller. In *Data Systems in Aerospace (DASIA99). ESA SP-447, Lisbon, Portugal*, pages 317–325, 1999.

[22] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing Timed Automata. TR CTIT 97-17, University of Twente, 1997. To appear in *Theoretical Computer Science*.