Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Automated Test Generation from Timed Automata

Brian Nielsen, Arne Skou

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

Abstract. Testing is the most dominating validation activity used by industry today, and there is an urgent need for improving its effectiveness, both with respect to the time and resources for test generation and execution, and obtained test coverage. We present a new technique for automatic generation of real-time black-box conformance tests for non-deterministic systems from a determinizable class of timed automata specifications with a dense time interpretation. In contrast to other attempts, our tests are generated using a coarse equivalence class partitioning of the specification. To analyze the specification, to synthesize the timed tests, and to guarantee coverage with respect to a coverage criterion, we use the efficient symbolic techniques recently developed for model checking of real-time systems. Application of our prototype tool to a realistic specification shows promising results in terms of both the test suite size, and the time and space used for test generation.

Key words: formal methods– real-time systems– automated testing– test case generation– testing tool– timed automata– symbolic analysis– test selection– domain testing– dense time–state partitioning– case study– performance evaluation– verification

1 Background

Communicating and embedded real-time systems remain among the most challenging class of systems to develop correctly. The challenge arises in part from the inherent complexity in these systems, but in particular from the lack of adequate methods and tools to deal with this complexity. This applies to most development activities, including specification, design and implementation.

Testing consists of executing a program or a physical system with the goal of finding errors. It is not unusual to spend more than a third of the total development time on testing in industrial projects, and it therefore constitutes a significant portion of the cost and the time to market of the product. Because testing is the most dominating validation activity used by industry today, there is an urgent need for improving its effectiveness, both with respect to the time and resources used for test generation and execution, and also the obtained coverage.

A potential improvement that is being examined by researchers is to make testing a formal method, and to provide tools that automate test case generation and execution. This approach has experienced some level of success: Formal specification and automatic test generation are being applied in practice [10,30,33,37], and academic as well as commercial test generations tools are emerging for specification languages like SDL (Specification and Design Language) [29,11,25,38], CSP (Communicating Sequential Processes) [34], LOTOS and PROMELA [21].

However, current tools do not address real-time systems, or only provide a limited support of testing the timing aspects. They often abstract away the actual time at which events are supplied or expected, or does not select these time instances thoroughly and systematically.

A large varitey of testing types and techniques exist, and they may be classified by the visibility of the implementation (black box or white box), the granularity of the implementation (function, component, system level), and by the correctness aspect being examined (robustness, reliability, performance, or behavior). This paper contributes with new techniques that will enable automatic generation of *black box* test cases that check the external *timing behavior* of real-time *systems* or components.

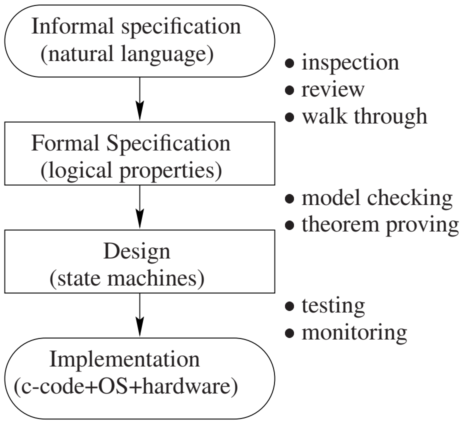


Fig. 1. Testing and verification are complementary techniques (inspired by Rushby [36]).

1.1 Model Based Systems Development

Testing is often perceived as an experimental and ad hoc approach to validating the correctness of a system, and it is also contrasted to *verification* which aims at formally proving system correctness. One kind of verification technology is *model checking* where a (fully) automatic tool examines whether the states of a behavioral description of a system satisfy a given property. Another approach is *theorem proving* where a proof assistant tool helps the system developer to prove that the system has a given property.

In our view, testing and verification are complementary techniques solving different problems. Therefore both should be used. The relation between different validation techniques and the (simplified) phases of software development is shown in Figure 1. Based on this observation we envision a development methodology where formal verification and testing are integral and complementary activities, both being centered around the use of formal and machine readable models.

The system analysts capture the informal requirements by interviewing customers, users, and domain experts about the requirements and expectations they have to the system. The requirements can then be formulated as logical properties or propositions. Completeness and soundness with respect to the informal requirements are achieved by reviews, inspections, and walk-throughs. Formal consistency check can be applied to the logical properties.

The analysis and design activities result in an abstract model or design of the system to be implemented: its components, the behavior of these components, and their interaction. The central or critical parts of the model can be specified in detail using a formal behavioral description language such as state machines or process algebra. The modelling and specification effort is by itself valuable because it gives very precise insight into the detailed operation of the system, but even more importantly, model checking can be used to ensure that this formal model satisfies the desired properties. Concistency check of the requirements, modeling, and model checking of the design are important activities, because they contribute to fault detection at the early development phases where correction can be made at much less cost that at the later phases.

Given a detailed model, the programmers implement the system using a particular programming language, compiler, and operating system. Automatic code generation from the formal model can also be used in special cases. However, with the current state of the art, system construction is largely an informal step carried out by humans. Errors may also be present in software components purchased from a third party, or in previously developed components used in the new product (and thereby in a different environment). Further, the target system may be faulty due to errors in the hardware or operating system, or the middleware. Therefore, the final physical system may be faulty even if its design is correct. It should also be mentioned here that many public safety boards require extensive testing of safety critical systems. It is insufficient, although important, to show that the design is correct: It must be demonstrated that the actual physical system is safe. However, the (verified) design or specification can now be used as a basis for (automatically) generating test cases.

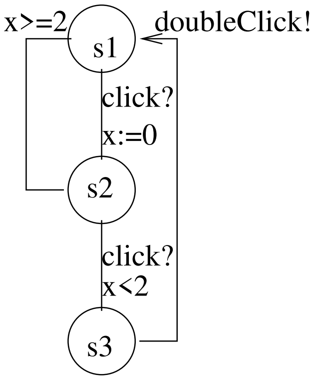
The fundamental *goal of testing*, which cannot be done by model checking, is to check if an actual running physical system of which we have no or incomplete knowledge conforms to a specification. A test case describes an experiment consisting of a sequence (or tree) of communications and associated verdicts to be executed on the implementation under test. The execution of a test results in a pass, fail, or an inconclusive verdict.

1.2 Using Formal Models for Testing

This section explains how a formal specficiation can be interpreted for test generation. Consider the timed automata specification of a mouse double-click detector in Figure 2a. A *timed automaton* is a finite state machine extended with continous clock variables, enabling conditions on edges, and clock resets, see [7].

In the initial state the automaton occupies location s_1, and clock x is zero. The value of clock x increases autonomously as time passes. When the user presses the mouse button (`click`), the automaton moves to location s_2 and simoultaneously resets clock x to zero. If the user clicks again before the clock x reaches a value of 2 time units, the automaton moves to location s_3, and then signals the detection of a double-click (`doubleClick`) after which it moves back to the initial location. If no click is made before 2 time units has elapsed after the first click, the automaton silently moves back to the initial state.

From this specification it is easy to suggest a collection of potential test cases that can be used to check the correctness of a given mouse double-click detector implementation. Some examples are shown in Figure 2b. The first is read as follows: First press click, wait 1.5 time units, press click, and if the



click!	click!	click!
wait(1.5)	wait(5)	wait(0.1)
click!	click!	click!
doubleClick?	doubleClick?	doubleClick?
(**pass**)	(**fail**)	(**pass**)

Fig. 2. Specification of a Mouse Double Click Detector, and potential test cases.

detector signals a double-click, the test execution should be given the verdict pass, i.e., it checks that double click is signaled when it should be. The second is read as: First press click, wait 5 time units, make the second click, and if the detector signals a double-click, the test execution should be given the verdict fail, because a double click is in fact not a legal behavior according to the specification, i.e., the test case checks that the detector only signals double click when permitted by the specification. The third test case checks that two fast clicks are detected as a double click.

Observe that an infinite number of test cases can be generated from this specification. There is an infinite number of delays to choose from between each observable event, and the loops of the specification can be unfolded arbitrarily many times. Good test selection strategies therefore become imperative.

Because test generation is based explicitly on a formal model, the generated tests will only be as good as the model. It is therefore important that the model itself has been checked by means of e.g., inspection, simulation or model checking.

1.3 Automated Testing

An overview of the setup for automated testing is depicted in Figure 3. Typically, a test generation tool inputs a specification formed by some kind of finite state machine description of the behavior that is required, desired, and forbidden by the implementation. A formalized *implementation relation* describes exactly what it means for an implementation to be correct with respect to a specification. The test generation tool interprets the specification or transforms it to a data structure appropriate for test generation, and then computes a set of test sequences. The test generator must select a subset of tests for execution only, because exhaustive testing is generally infeasible. Test selection can be based on manually stated test purposes, a fault model, or on a coverage criterion of the specification or implementation.

The generated test cases are usually abstract and given at the same level of abstraction as the specification. To become executable, the test cases must be interpreted by a test execution tool. The translation of abstract events into concrete events involves completing parameter lists, and encoding of these to bits strings that can be fed to the implementation under test. The execution tool communicates the concrete events to and from the implementation using the programming interface that is available in the test execution environment. Further, it manages timers, monitors the implementation under test, maintains log files, controls the overall progress of test execution, and assigns verdicts to the test execution.

Sometimes, test suites are not generated explicitly and stored in files, but are rather generated event by event as it is beeing executed. This approach is known as *on-the-fly* test generation and execution.

1.4 Testing Real-Time Systems

To test real-time systems, the specification language must be extended with constructs for expressing real-time constraints, the implementation relation must be generalized to consider the temporal dimension, and the data structures and algorithms used to generate tests must be revised to operate on a potentially infinite set of states. Further, the test selection problem is worsened because a huge number of time instances are relevant to test. It is therefore necessary to make good decisions of *when* to deliver an input to the system, and *when* to expect an output. The time dimension must be tested thoroughly and systematically because real-time systems are often safety critical. Automated test generation for real-time systems is a fairly new research area, and only few proposals exist that deal with these problems.

This paper presents a new technique for automatic generation of timed tests from a restricted class of dense timed automata specifications. We permit both non-deterministic specifications and (black-box) implementations. Our implementation relation is therefore based on Hennessy's classical testing theory [31] for concurrent systems, which we have generalized to take time into account.

We believe that it is important to allow non-deterministic specifications and implementations. Implementations of real-time systems are usually (indeterminate) concurrent systems, and consequently, the tester cannot in general control or know the order in which events are processed by the implementation. Further, the tester may not always be able to control every parameter of the operating environment of the implementation, such as the temperature, and must be prepared to accept the output delivered. Non-determinism is often used is specifications to abstract over internal descisions in the implementation which may not be known, or is too complicated for formal modelling. Specifically for real-time systems, the

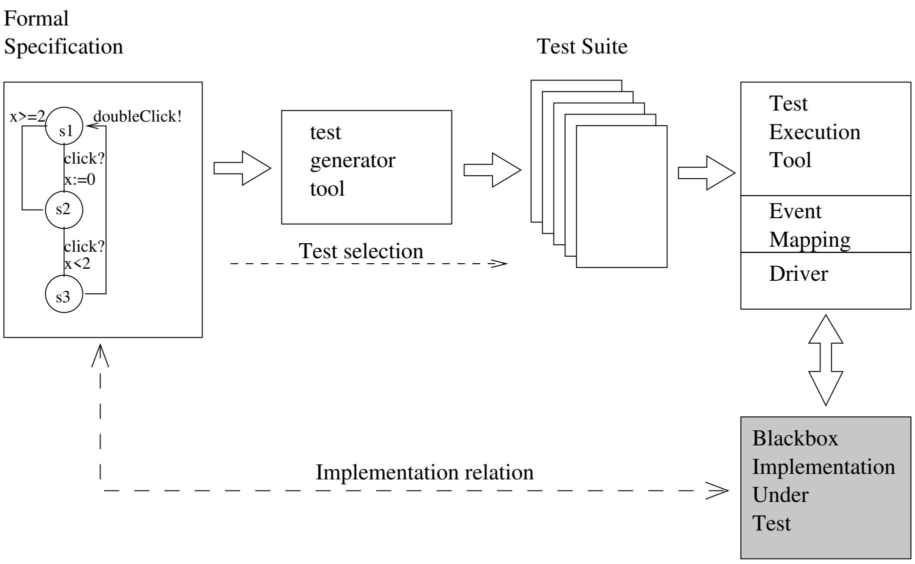


Fig. 3. Automated Testing.

response times of the implementation may vary, and the tester must be prepared for the output in a bounded interval, rather than at specific timepoints.

We propose to select test cases by partitioning the state space into coarse grained equivalence classes that in a systematic way preserve essential timing and deadlock information, and select a few tests for each class. This approach is inspired by sequential black-box testing techniques frequently referred to as domain- or partition testing [4], by regarding the clocks of a timed specification as (oddly behaving) input parameters.

We present an algorithm and data structure for systematically generating timed Hennessy tests. The algorithm ensures that the specification will be covered such that the relevant Hennessy tests for each reachable equivalence class will be generated. To compute and cover the reachable equivalence classes, and to compute the timed test sequences, we employ efficient symbolic reachability techniques based on constraint solving that have recently been developed for model checking of timed automata [22,9,41,6,27].

1.5 Contributions

In summary, the contributions of the paper are:

- We propose a *coarse* equivalence class partitioning of the state space and use this for *automatic* test selection.
- Other work on test generation for real-time systems allows deterministic specifications only, and use trace inclusion as implementation relation. We permit both *non-deterministic* specifications and (black-box) implementations, and use an implementation relation based on Hennessys testing theory that takes *deadlocks* into account.
- Application of the recently developed *symbolic reachability techniques* has to our knowledge not previously been applied to test generation.
- Our techniques are implemented in a prototype *test generation tool*, RTCAT.
- We apply our technique to a realistic case study, the Philips Audio Protocol, and evaluate the generated tests qualitatively as well as quantitatively.
- We provide *experimental data* about the efficiency of our technique. Application of RTCAT to one small and one larger case study results in encouragingly small test suites.

The remainder of the paper is organized as follows. Section 2 summarizes the related work. Section 3 introduces Hennessy tests, the specification language, the symbolic reachability methods, and test selection strategy. Section 4 presents the test generation algorithm, and Section 5 the facilities of our tool. Section 6 describes the case study, whereas Section 7 contains our experimental results. Finally, Section 8 concludes the paper and suggests future work.

2 Related Work

Springintveld et al. proved in [39] that *exhaustive* testing wrt. *trace equivalence* of *deterministic* timed automata with a *dense time* interpretation is theoretically possible, but highly infeasible in practice. Exhaustive here means that every incorrect implementation would be rejected by some test case. Another result generating checking sequences for a discretized *deterministic* timed automaton is presented by En-Nouaary et al. in [23]. Although the required discretization step size $(1/(|X|+2)$, where $|X|$ is the number of clocks) in [23] is more reasonable than [39], it still appears to be too small for most practical applications because too many tests are generated. Both of these techniques are based on computing checking sequences from the so called *region* graph technique due to Alur and Dill[1]. Clock regions are very fine-grained equivalence classes of clock valuations. We argue that coarser partitions are needed in practice. Further, our equivalence class partitioning as well as the used symbolic techniques are much less sensitive to the clock constants and the number of clocks appearing in the specification compared to the region construct.

Cardell-Oliver and Glover showed in [15] how to derive checking sequence from a *discrete time*, *deterministic*, timed transition system model. Their approach is implemented in a tool that is applied to a series of small cases. Their result indicates that the approach is feasible, at least for small systems, but problems arise if the implementation has more states than the specification. No test selection wrt. the time dimension is performed, i.e., an action is taken at all the time instances it is enabled. This work has recently been carried to a restricted timed automata model [14] in which a discretization step of 1 suffices. Observe that this cannot ensure that equivalence classes as proposed in this paper can be covered by a test. Moreover, it does not present an effective solution to the test selection problem, and the complexity remains highly dependent on the clock constants that appear in the specification.

Clarke and Lee [17, 18] also propose domain testing 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. Their technique appear to produce much fewer tests than region based generation. The time requirements are specified as directed acyclic graphs called *constraint graphs*. Compared to timed automata this specification language appear very restricted, e.g., because their constraint graphs must be acyclic this only permits specification of finite behaviors. Their domains are "nice" linear intervals that are directly available in the constraint graph. In our work they are (convex) polyhedra of a dimension equal to the number of clocks.

Braberman et al. [12] describe an approach where a structured analysis/structured design real-time model is represented as a timed Petri net. Analysis methods for timed Petri nets based on constraint solving can be used to generate a symbolic *timed reachability tree* up to a predefined time bound. From this, specific timed test sequences can be chosen. This work shares with ours the generation of tests from a symbolic representation of the state space. We *guarantee coverage* according to a well defined criterion without reference to a predefined or explicitly given upper time bound. The paper also proposes other selection criteria, mostly based on the type and order of the events in the trace. However, they are concerned with generating *traces only*, and not on deadlock properties as we are. The paper describes no specific data structures or algorithms for constraint solving, and states no results regarding their efficiency. Their approach does not appear to be implemented.

Castanet et al. presents in [16] an approach where timed test *traces* can be generated from timed automata specifications. Test selection must be done *manually* through engineerer 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. Further, the test sequences appear to be synthesized from paths available directly in an intermediate timed automaton formed by a synchronous product of the specification and the test purpose, and not from a (symbolic) interpretation thereof. This approach therefore risks generating tests that need not be passed by the implementation, or not finding a test satisfying the test purpose when one in fact exists.

Finally, test generation from a discrete time temporal logic is investigated by [30].

3 Preliminaries

3.1 Hennessy Tests

In Hennessy's testing theory [31] specifications $\mathcal{S}$ are defined as finite state labelled transition systems over a given finite set of actions Act. Also, it assumes that implementations $\mathcal{I}$ (and specifications) can be observed by finite tests $\mathcal{T}$ via a sequence of synchronous CCS-like communications. The implementation under test is composed in parallel (using a CCS-like parallel composition operator $\|$) with a process modelling the tester, and consequently, the execution of a test consists of a finite sequence of communications forming a so–called *computation* — denoted by $Comp(\mathcal{T} \parallel \mathcal{I})$ (or $Comp(\mathcal{T} \parallel \mathcal{S})$). A test execution is assigned a verdict (pass, fail or inconclusive), and a computation is *successful* if it terminates after an observation having the verdict pass.

Hennessy tests have the following abstract syntax $\mathcal{L}_{\text{tlts}}$: (1) $\mathbf{after}\ \sigma\ \mathbf{must}\ A$, (2) $\mathbf{can}\ \sigma$, and (3) $\mathbf{after}\ \sigma\ \mathbf{must}\ \emptyset$, where $\sigma \in Act^*$ and $A \subseteq Act$. Informally, (1) is successful if at least one of the observations in A (called a *must set*) can be observed whenever the trace σ is observed, (2) is successful if σ is a prefix of the observed system, and (3) is successful if this is not the case (i.e. σ is not a prefix).

The basic idea in Hennessy's testing theory is to compare systems based on the tests they pass: Two systems are regarded as equivalent if no experimenter can distinguish them by executing tests. Hennessy's testing equivalence thus requires that the specification and implementation exactly passes the same tests. Similarly, the testing preorder requires that the implementation passes at least the same tests as the specification.

Because some computations of a test may yield success and some may not, there are two possible definitions of passing a test. One is that an implementation passes a test if the possible executions *may* report success, i.e., at least one of the possible computations is required to report success. The other choice

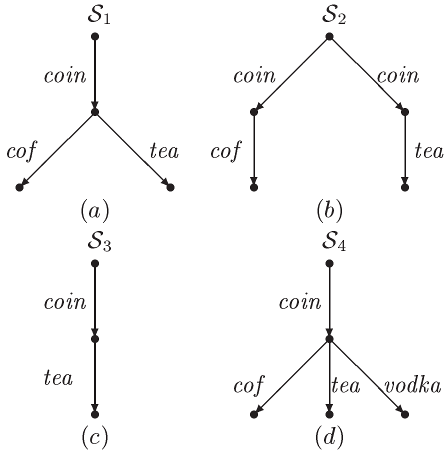


Fig. 5. Four different coffe machines modelled as labelled transition systems.

is that an implementation passes a test if the possible executions *must* report success, i.e., all computations are required to report success.

The must (may) preorder requires that every test that must (may) be passed by the specification must (may) also be passed by the implementation. In non-deterministic systems these notions do not coincide. The testing preorder defined formally in Definition 4 requires satisfaction on both the must and may preorders.

Definition 4. *The Testing Preorder* $\sqsubseteq_{\mathbf{te}}$:

1. $\mathcal{S}$ **must** $\mathcal{T}$ *iff* $\forall \Sigma \in Comp(\mathcal{T} \parallel \mathcal{S})$. Σ is successful.
2. $\mathcal{S}$ **may** $\mathcal{T}$ *iff* $\exists \Sigma \in Comp(\mathcal{T} \parallel \mathcal{S})$. Σ is successful.
3. $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{\mathrm{tlts}}$. $\mathcal{S}$ **must** $\mathcal{T}$ *implies* $\mathcal{I}$ **must** $\mathcal{T}$
4. $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{\mathrm{tlts}}$. $\mathcal{S}$ **may** $\mathcal{T}$ *implies* $\mathcal{I}$ **may** $\mathcal{T}$
5. $\mathcal{S} \sqsubseteq_{\mathbf{te}} \mathcal{I}$ *iff* $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ and $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$

□

A further relation called **conf** proposed by Brinksma [13] is commonly used in the field of protocol conformance testing. It is similar to the must preorder except that it only considers must tests with traces in the specification, i.e., tests of the form **after** σ **must** A with $\sigma \in Tr(\mathcal{S})$.

The examples in Figure 5 compare the merits of the different implementation relations, and illustrate that the importance of making the right choice.

First compare the vending machine specifications $\mathcal{S}_1$ and $\mathcal{S}_2$. The specification $\mathcal{S}_1$ requires that an implementation after accepting a coin enables the environment to choose between tea and coffee. $\mathcal{S}_2$ would be a faulty implementation using either the conformance or the must preorder, because it chooses internally whether to enable tea or coffee. $\mathcal{S}_1$ and $\mathcal{S}_2$ are distinguishable by the test **after** $coin$ **must** $\{cof\}$ that $\mathcal{S}_1$ always passes, but $\mathcal{S}_2$ does not. However, $\mathcal{S}_1$ and $\mathcal{S}_2$ are indistinguishable by the may preorder.

$\mathcal{S}_2$ can be interpreted as a specification that allows the implementor to choose between implementing a coffee or a tea machine (or both). $\mathcal{S}_3$ is a legal implementation of $\mathcal{S}_2$ according to the must and conformance preorder, but not according to the may preorder: $\mathcal{S}_2$ **may** $coin \cdot cof$ passed by $\mathcal{S}_2$ but never by $\mathcal{S}_3$. The may preorder ensures that every behavior that can be exhibited by the specification can also be exhibited by the implementation, i.e., it checks the robustness or liveness aspects of the implementation.

$\mathcal{S}_4$ is a deterministic alternative that does not have the same problem as $\mathcal{S}_2$. However, it is able to sell vodka in addition to tea and coffee. If the goal is to ensure that we can buy coffee or tea, thus using the conformance relation, we can safely accept $\mathcal{S}_4$ as an implementation, i.e, $\mathcal{S}_4$ **conf** $\mathcal{S}_1$. However, such extended functionality may be catastrophic e.g., selling alcoholic drinks to minors, or splashing coffee on the floor when no coins and cups have been inserted. The must preorder does not permit such added behavior, as demonstrated by the test **after** $coin \cdot vodka$ **must** $\emptyset$ that $\mathcal{S}_1$ passes, but not $\mathcal{S}_4$.

A must test **after** σ **must** A can be generated from a specification by 1) finding a trace σ in the specification, 2) computing the states that are reachable after that trace, and 3) computing a set of actions A that must be accepted in these states. To facilitate and ease systematic generation of all relevant tests, the specification can be converted to a success graph (or acceptance graph [19]) data structure. A success graph is a *deterministic* state machine trace equivalent to the specification, and whose nodes are labeled with the must sets holding in that node, the set of actions that are possible, and the actions that must be refused.

We propose a simple timed generalization of Hennessy's tests. In a timed test **after** σ **must** A (or **after** σ **must** $\emptyset$), σ becomes a timed trace (a sequence of alternating actions and time delays), after which an action in A must be accepted immediately. Similarly, a test **can** σ (**after** σ **must** $\emptyset$) becomes a timed trace satisfied if σ is (is not) a prefix trace of the observed system. A test will be modelled by an executable timed automaton whose locations are labelled with pass, fail, or inconclusive verdicts.

3.2 Event Recording Automata

Two of the surprising undecidability results from the theoretical work on timed languages described by timed automata is 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. Also, internal actions increase the expressiveness of timed automata, and specifically it has been shown that internal actions with clock resets on cycles cannot be removed [40]. The Event Recording Automata model (ERA) was proposed by Alur, Fix, and Henzinger in

[2] as a simple, clean subclass of timed automata that is determinizable and has language inclusion as a decidable property.

Definition 6. *Event Recording Automaton*:

1. Let X be set of real-valued clocks. The clock constraints (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 $x_1, x_2 \in X$.
2. 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. $X = \{x_a \mid a \in Act\}$ is the set of real-valued clocks. We use the term *location* to denote a node in the automaton, and reserve the term *state* to denote the semantic state of the automaton also including clock values. We write $l \xrightarrow{g,a} l'$ when $(l, g, a, l') \in E$.

□

Like a timed automaton, an ERA has a set of clocks that can be used in guards on actions, and that 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]. We shall finally also assume that all observable actions are *urgent* meaning that synchronization between the environment and automaton takes place immediately when the parties have enabled a pair of complementary actions. With non-urgent observable actions this synchronization delay would be unbounded.

Figure 7 shows an example of a small ERA. It models a coffee vending machine built for impatient users such as busy researchers. When the user has inserted a coin (`coin`), he must press the give button (`give`) to indicate how eager he is to get a drink. If he is very eager, he presses `give` soon after inserting the coin, and the vending machine outputs thin coffee (`thinCof`); apparently, there is insufficient time to brew good coffee. If he waits more than four time units, he is certain to get good coffee (`cof`). If he presses `give` after exactly four time units, the outcome is non-deterministic.

In a *deterministic* timed automata, the choice of the next edge to be taken is uniquely determined by the automaton's current location, the input action, and the time the input event is offered. The determinization procedure for ERAs is given by [2], and is conceptually a simple extension of the usual subset construction used in the untimed case, only now the guards must be taken into account. Figure 8 illustrates the technique. Observe how the guards of the `give` edges from $\{s2\}$ become mutually exclusive such that either both are enabled, or only one of them is.

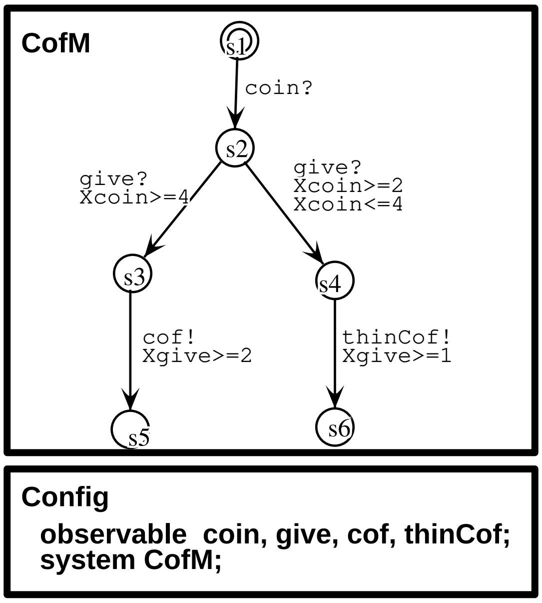


Fig. 7. ERA specification of a coffee vending machine.

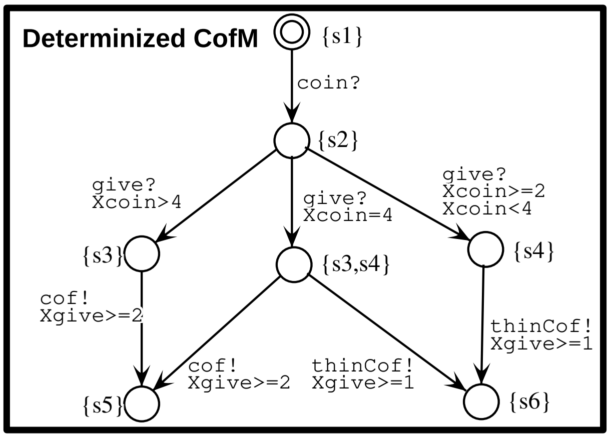


Fig. 8. Determinized coffee vending machine (b).

3.3 Symbolic Representation

Timed automata with a dense time interpretation cannot be analyzed by finite state techniques, but must rather be analyzed symbolically. Efficient symbolic reachability techniques have been developed for model checking of timed automata [22,9,41,6,27]. Specifically, we shall employ similar techniques as those developed for the UPPAAL tool [41,6,27].

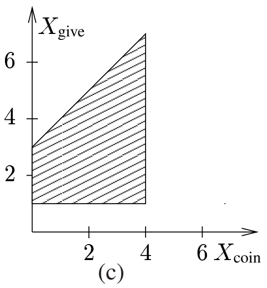
The state of a timed automaton can be represented by the pair $\langle \bar{l}, \bar{u} \rangle$, where $\bar{l}$ is the automaton's current location (vector), and where $\bar{u}$ is the vector of its current clock values. A *zone* z is a conjunction of clock constraints of the form $x_1 \sim c$ or $x_1 - x_2 \sim c$ with $\sim \in \{\leq, <, =, >, \geq\}$, or equivalently, the solution set to these constraints. Viewed graphically, the solution set of a zone forms a convex polyhedron of a number of dimensions corresponding to the number of

$$\langle s_2, \begin{smallmatrix} X_{\text{coin}} = 2.00 \\ X_{\text{give}} = 3.14 \end{smallmatrix} \rangle$$

(a)

$$\begin{aligned} &[s2, z], \\ &z = X_{\text{give}} \geq 1 \\ &\quad \wedge\ X_{\text{coin}} \leq 4 \\ &\quad \wedge\ X_{\text{give}} - X_{\text{coin}} \leq 3 \end{aligned}$$

(b)



(c)

	0	X_{coin}	X_{give}
0	$\times$	4	7
X_{coin}	0	$\times$	3
X_{give}	-1	3	$\times$

(d)

Fig. 9. State of a timed automaton (a), a symbolic state (b), a polyhedron containing the solution set to zone z, and DBM for zone z (d).

clocks. A symbolic state $[\bar{l}, z]$ represents a (infinite) set of states: $\{\langle \bar{l}, \bar{u} \rangle \mid \bar{u} \in z\}$, see the example in Figure 9.

Zones can be represented and manipulated efficiently by the *difference bound matrix* (DBM) data structure[5]. DBMs were first applied to represent clock differences by Dill in [22]. A DBM represents clock difference constraints of the form $x_i - x_j \prec c_{ij}$ by a $(n+1) \times (n+1)$ matrix where n is the number of clocks, and $\prec \in \{\leq, <\}$.

An efficient set of operations on zones allows the following to be computed:

- The symbolic state that results by taking an edge from a given source symbolic state can be computed.
- The reachable state space can be computed. Forward reachability analysis starts in the initial state $(\bar{l}_0, \bar{0})$, and computes the symbolic states that can be reached by executing an action from an existing one, or by letting time pass. When a new symbolic state is included in one previously visited, no further exploration of the new state needs to take place. Forward reachability thus terminates when no new states can be reached.
- Given a symbolic path to a symbolic state, a concrete timed trace leading to it (or subset thereof) can be computed by back propagating its constraints along the symbolic path used to reach it, and by choosing specific time points along this trace.

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.

3.4 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 only pass through the states 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 that 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 interior 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, because this may detect errors where a timer times out prematurely.

3.5 Domain Based Test Selection

It is important to systematically select and generate a limited amount of tests, because exhaustive testing is generally infeasible. 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. In industrial projects it is highly desirable that there is such a well defined metric of the testing thoroughness, and that this can be measured.

As previously stated, our approach is inspired by sequential black-box testing techniques frequently referred to as domain- or partition testing [4]. We regard the clocks of a timed specification as (oddly behaving) input parameters.

To exemplify the analogy, consider the 'maxPositive' function specified below to return the maximum positive value of two arbitrary integers, x and y. If both arguments are less than zero the function is to return zero:

domain	condition	expected output
1	$x < 0 \wedge x > y$	0
2	$y < 0 \wedge x \leq y$	0
3	$x \geq 0 \wedge x > y$	x
4	$y \geq 0 \wedge x \leq y$	y

Table 10. Proposed domains for maxPositive function

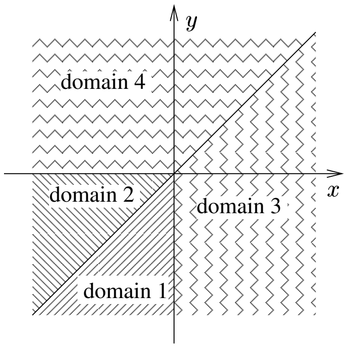


Fig. 11. Visualization of proposed domains for the maxPositive function.

$$\text{maxPositive}(x, y) =_{\text{def}} max(0, max(x, y))$$

$$max =_{\text{def}} \begin{cases} x, \text{ if } x > y \\ y, \text{ otherwise} \end{cases}$$

Obviously, this function cannot be tested with all possible pairs of integers as inputs. A systematic strategy for dealing with this problem is to partition the input variables into input *domains*, i.e., sets of inputs that the program is expected to treat "identically" (e.g., pass through the same program path), and choose only a few representatives from each domain.

For the 'maxPositive' example, one might use four domains, one for each of the four cases in the specification. The resulting domains, tabulated in Table 10, are described as inequations of x and y whose solution set is depicted in Figure 11. Test input data can be derived from the inequations that define each domain. Concluding correctness from testing with only a few representatives requires a uniformity hypothesis stating that if the implementation is correct for one input in the domains, it is correct for all. To support this hypothesis boundary value analysis is applied by selecting several extreme values in the domain. Thus, at least four test cases should be generated, but usually several interior and extreme values are chosen.

For real-time systems we propose to partition the clock valuations into domains and ensure that each such domain is tested systematically. Similarly, the concept of extreme values carries over to the time domain to mean extreme clock valuations close to the borders of clock value domains. However, observe that clock values are not parameters that can be directly supplied to a function, but can only be controlled implicitly by stimulating the target system by a timed trace of events.

4 A Test Generation Algorithm

Our equivalence class partitioning and coverage criterion are introduced intuitively in Section 4.1. The equivalence class partitioning is defined formally in Section 4.2. An algorithm for constructing the equivalence classes of a specification is provided in Section 4.3. The test generation algorithm is presented in Section 4.4.

4.1 Selection Criterion

The implementation relation defined in Definition 4 defines what test cases are necessary and what structure the must have: Only those Hennessy tests passed by the specification are relevant and need be generated and executed. However, even with this restriction exhaustive testing requires an infinite number of test cases, and no test case is in theory expendable. In practice there is only resources to execute a finite and very limited subset, and consequently, test selection must be employed to compute an executable subset.

We propose a selection criterion based on partitioning the state space of the specification into coarse equivalence classes, and requiring 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. These observations are used to increase the confidence that the equivalence classes are correctly implemented. The partitioning and observations can be done in numerous ways, and some options are explored and formally defined in [32].

Given the partitioning stated in the following, the *stable edge set criterion* implemented in RTCAT requires that all relevant *simple deadlock* observations of the forms $\mathbf{after}\ \epsilon\ \mathbf{must}\ A$ (a *must* property), $\mathbf{after}\ a\ \mathbf{must}\ \emptyset$ (also called $\mathbf{cannot}\ a$, a *refusal* property), and $\mathbf{can}\ a$ (a *may* property) are made at least once in each class. These properties are derived from the correctness criterion, the testing preorder.

Consider the (determinized) coffee vending machine from Figure 8. We propose to partition the clock valuations into domains, e.g., for location $\{s3, s4\}$ we argue that there are 3 important cases: domain 1 where neither of the outgoing edges are enabled, domain 2 where the `thinCof` edge is enabled, but not the `cof` edge, and domain 3 where both are enabled, see Figure 12.

The idea behind this partitioning is that the specification remains stable with respect to the locations (recall that it may be non-deterministic) it might occupy, and edges that might be enabled; only the clock values are changing. The hypothesis is that a correct implementation is also stable because no new actions need to be enabled or disabled to match the behavior of the specification. In contrast, when its locations or enabled edges change, some internal action (e.g a clock interrupt) must trigger the implementation causing it to change state, and hence, its behavior should be reexamined.

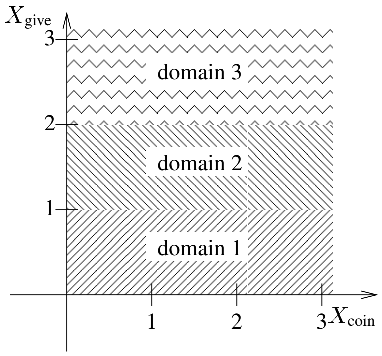


Fig. 12. Visualization of the proposed clock valuation domains for the coffee vending machine location $\{s3, s4\}$.

$[\{s3, s4\}, p_5]$ $p_5 : Xgive \geq 2$	$[\{s3, s4\}, p_6]$ $p_6 : Xgive \in [1, 2)$	$[\{s3, s4\}, p_7]$ $p_7 : Xgive < 1$
cannot coin **cannot** give **can** cof **can** thinCof **must**{ cof, thinCof}	**cannot** coin **cannot** cof **cannot** give **can** thinCof	**cannot** coin **cannot** cof **cannot** give **cannot** thinCof

Fig. 13. 3 Examples of Decorated Equivalence Classes (Coffee Machine)

In general we propose an equivalence class for each subset of edges possibly enabled from each subset of locations. Each such class is decorated with the simple deadlock observations satisfied in that class, see Figure 13. A test case consists of a timed trace leading to a desired state in an equivalence class followed by one of the simple deadlock observations. Suppose the implementation erroneously allowed coffee to be produced immediately after a `coin` was entered and `give` was pushed 4 after time units. The specification would then occupy class $[\{s3, s4\}, Xgive < 1]$ that satisfies the property $\mathbf{cannot}\ cof$, see Figure 13. The coverage criterion guarantees that a test is generated that could detect this (particular) error.

In general there is a tradeoff between partition size, error detection capability, the strength of the required uniformity hypothesis, and the number of required test cases, and thus the cost of the test suite. The finer the partitioning, the higher cost of generating and executing the test suite. Our 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 6.

A finer partitioning like covering the regions of a specification results in an increased timewise resolution of the test suite, i.e., the edges will be visited with many more (timed) test points, thus increasing the likelihood of detecting erroneously implemented guards, but it is important to note that it does not examine more actions or edges than ours, because our reachability analysis guarantees that every (reachable) action or edge will be covered by at least one test. Covering the region graph also appear to be extremely costly in terms of the number of required tests both in theory [39] and in practice [23].

Our partitioning has the nice formal property that the states in the same equivalence class are also equivalent with respect the previously stated *simple deadlock* properties. This follows from the absence of τ actions, and the fact that only enabled edges, and not the precise clock values, affects the satisfaction of these properties. In contrast, different equivalence classes typically satisfy different simple deadlock properties. It is therefore natural to check that the implementation matches these properties for each equivalence class. Using an even coarser partitioning is therefore likely to leave out significant timing and deadlock behavior.

Our approach provides a heuristic that guarantees that a well defined set of interesting scenarios in the specification has been automatically, and completely and systematically explored.

4.2 *State Partitioning*

From each control location L (recall that a location in a determinized automaton is the set of locations of the original automaton that the automaton can possibly occupy after a given trace), the clock valuations are partitioned such that two clock valuations belong to the same equivalence class iff they enable precisely the same edges from L, i.e. the states are equivalent wrt. the enabled edges. An equivalence class will be represented by a pair $[L, p]$, where L is a set of location vectors, and p is the inequation describing the clock constraints that must hold for that class, i.e., $[L, p]$ is the set of states $\{\langle L, \bar{u}\rangle \mid \bar{u} \in p\}$. Further, to obtain equivalence classes that are contiguous convex polyhedra, and to reuse the existing efficient symbolic techniques, this constraint is rewritten to its disjunctive normal form. Each disjunct is treated as its own equivalence class. The partitioning from a given set of locations is defined formally in Definition 14.

Definition 14. *State partitioning* $\Psi(L)$:
Let L be a set of location vectors, $E(L)$ the set of edges starting in a location vector in L, E a set of edges, and $\Gamma(E) = \{g \mid \bar{l} \xrightarrow{g,a} \bar{l}' \in E\}$. Recall from Definition 6 that $G(X)$ denotes the guards generated by the syntax $g ::= \gamma \mid g \wedge g$ where γ is a basic clock constraint of the form $x_1 \sim c$ or $x_1 - x_2 \sim c$.

Let P be a constraint over clock inequations γ composed using *any* of the logical connectives $\wedge, \vee$, or $\neg$. Let $\mathrm{DNF}(P)$ denote a function that rewrites constraint P to its equivalent disjunctive normal form, i.e., such that $\bigvee_i \bigwedge_j \gamma_{ij} = P$. Each conjunct in the disjunctive form can be written as a guard g in $G(X)$ by appropriately negating basic clock constraints where required. The disjunctive normal form can therefore be interpreted as a disjunction of guards such that $\bigvee_i g_i = \bigvee_i \bigwedge_j \gamma_{ij}$. The *set* of guards g_i whose disjunction equals the disjunctive normal form is denoted GDNF, i.e, $\mathrm{GDNF}(P_E) = \{g_i \in G(X) \mid \bigvee_i g_i = \mathrm{DNF}(P_E)\}$.

1. $\Psi(L) = \{P_E \mid E \in 2^{E(L)}\}$, where
   $P_E = \bigwedge_{g \in \Gamma(E)} g \wedge \bigwedge_{g \in \Gamma(E(L)-E)} \neg g$
2. $\Psi_{\mathrm{dnf}}(L) = \bigcup_{P_E \in \Psi(L)} \mathrm{GDNF}(P_E)$

□

The equivalence classes for the coffee vending machine is depicted in Figure 17. Each equivalence class $[L, p]$ can now be decorated with the action sets M, C, R defined in Definition 15. Some examples are shown in Figure 13.

Definition 15. *Decorated Equivalence Classes*:
Define $\mathrm{Sort}([L, p]) = \{a \mid \exists \langle L, \bar{u}\rangle \in [L, p].\ \langle L, \bar{u}\rangle \xrightarrow{a}\}$, and $\mathrm{Must}([L, p]) = \{A \mid \exists \langle L, \bar{u}\rangle \in [L, p].\ \langle L, \bar{u}\rangle \models$ **after** ϵ **must** $A\}$

1. $M([L, p]) = \mathrm{Must}([L, p])$.
2. $C([L, p]) = \mathrm{Sort}([L, p])$.
3. $R([L, p]) = \mathit{Act} - \mathrm{Sort}([L, p])$.

□

M contains the sets of actions necessary to generate the must tests, C the may tests, and R the refusal tests for that class. Specifically, if σ is a timed trace leading to class $[L, p]$, and $A \in M([L, p])$ then **after** σ **must** A is a test to be passed for that class. So is **after** $\sigma \cdot a$ **must** $\emptyset$ if $a \in R([L, p])$, and **can** $\sigma \cdot a$ if $a \in C([L, p])$. The number of generated tests can be further reduced by removing tests that are logically passed by another test, i.e., the must sets can be reduced to $M([L, p]) = \mathit{min}_{\subseteq}\mathrm{Must}([L, p])$ (where $\mathit{min}_{\subseteq}(M)$ gives the set of minimal elements of M under subset inclusion), and the actions observed during the execution of a must test can be removed from the may tests, i.e., $C([L, p]) = \mathrm{Sort}([L, p]) - \bigcup_{A \in M([L,p])} A$.

4.3 Equivalence Class Graph Construction

We view the state space of the specification as a graph of equivalence classes. A node in this graph contains an equivalence class. An edge between two nodes is labeled with an observable action, and represents the possibility of executing an action in a state in the source node, waiting some amount of time, and thereby entering a state in the target node. The graph is constructed by starting from an existing node $[L, p]$ (initially the equivalence classes of the initial location), and then for each enabled action a, by computing the set of locations L' that can be entered by executing the a action from the equivalence class. Then the partitions p' of location L' can be computed according to Definition 14 (2). Every $[L', p']$ is then an a successor of $[L, p]$. It should be noted that only equivalence classes whose constraints have solutions need to be represented. The equivalence class graph is defined inductively in Definition 16. This definition can easily be turned into an algorithm for constructing the equivalence class graph.

Definition 16. *Equivalence Class Graph*:
The nodes and edges are defined inductively as:

1. The set $\{[L_0, p] \mid L_0 = \{\bar{l}_0\},\ p \in \Psi_{\mathrm{dnf}}(L_0),$ and $p \neq \emptyset\}$ are nodes.
2. if $[L, p]$ is a node, so is $[L', p']$, and $[L, p] \xrightarrow{a} [L', p']$ is an edge if $p' \neq \emptyset$, where
   $L' = \{\bar{l}' \mid \exists \bar{l} \in L.\ \bar{l} \xrightarrow{g,a} \bar{l}'\}$, and $p' \in \Psi_{\mathrm{dnf}}(L')$.

□

The construction algorithm implicitly determinizes the specification, but preserves the non-determinism of the original specification. The equivalence class graph preserves all timed traces of the specification, and furthermore preserves the required deadlock information for our timed Hennessy tests of the specification by the M, C, and R action sets stored in each node. The non-determinism found in the original specification is therefore maintained, but is represented differently, and in a way that is more convenient for test generation: A test is composed of a trace, a deadlock observation possible in the specification thereafter, and associated verdicts, and this information can be found simply by following a path in the equivalence class graph. All timed Hennessy tests that the specification passes can thus be generated from this graph. The explicit graph also makes it easy to ensure coverage according to the coverage criterion by marking the visited parts of the graph during test generation. Also note that the determinization also allows the equivalence classes (relatively) efficiently; there is no need to form every subset of edges in the entire specification, only those from the same location in the determinized automaton. The equivalence class graph for the coffee machine is depicted in Figure 17.

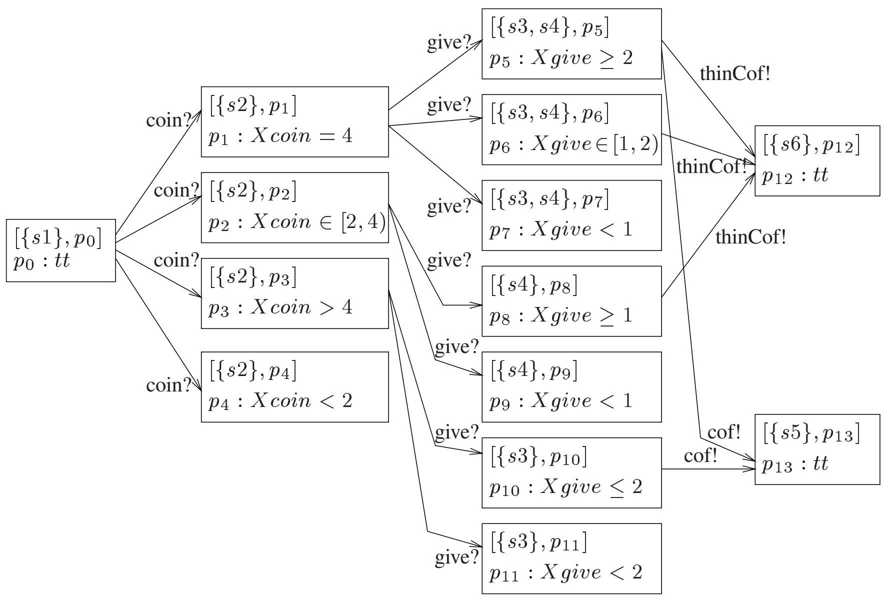


Fig. 17. Equivalence class graph for the coffee machine.

4.4 Overall Algorithm

The equivalence class graph preserved the necessary information for generating timed Hennessy tests. However, it also contains behavior and states *not* found in the specification, and using such behavior will result in irrelevant and unsound tests. An unsound test may produce the verdict fail even when the implementation conforms to the specification. According to the testing preorder only tests passed by the specification should be generated. To ensure soundness, only the traces and deadlock properties actually contained in the specification may be used in a generated test. To find these, we therefore interpret the specification symbolically, and generate the timed Hennessy tests from a representation of only the reachable states and behavior.

For instance, no test can be constructed that must be passed by the specification that checks the behavior of the coffee vending machine in state $\langle\{s3, s4\}, (X_{\text{coin}} = 2, X_{\text{give}} = 2)\rangle$, because whenever the automaton occupies location $\{s3, s4\}$, clock X_{coin} is at least 4. Similarly, it may be necessary to unfold loops in the equivalence class graph several time to ensure that all reachable classes are visited by a test. An example of this is provided in Section 6.1.

Moreover, the use of reachability analysis gives a termination criterion for the symbolic interpretation; when completed it guarantees that every reachable equivalence class is represented by some symbolic state. Thus, we are able to guarantee coverage by inspecting the reached symbolic states.

Algorithm 18 presents the main steps of our generation procedure. Step 1 constructs the equivalence class graph as described in Section 4.2. The result of step 2 is a *symbolic reachability graph*. Nodes in this graph consist of symbolic states $[L, z/p]$ where L is a set of location vectors, and where z is a constraint characterizing a set of reachable clock valuations also in p, i.e., $z \subseteq p$. An edge represents that the target state is reachable by executing an action from the source state and then waiting some amount of time.

The nodes in the reachability graph are decorated according to Definition 15 in step 3. Step 4 initializes an empty set that contains the symbolic states from which tests have be generated so far.

Step 5 contains the generation process itself. Note that the same partition may be traversed many times during forward reachability analysis. The coverage criterion only requires that tests are generated from one point of the partition. Algorithm 18 therefore only generates test for the first symbolic state that reaches a given partition, and uses the set *Tested* to ignore subsequent passes over the same partition. This ensures that all the may, must, and refusal properties are only generated once per partition, and thus reduces the number of produced test cases. Other strategies such as testing all reached symbolic states, or only testing certain designated locations deemed critical by the user, can easily be implemented.

Algorithm 18. *Overall Test Case Generation Algorithm*:

input: ERA specification $\mathcal{S}$.
output: A covering set of relevant timed Hennessy properties.

1. Compute $\mathcal{S}_p$ = Stable Edge Set Partition Graph$(\mathcal{S})$.
2. Compute $\mathcal{S}_r$ = Reachability$(\mathcal{S}_p)$.
3. Label every $[L, z/p] \in \mathcal{S}_r$ with the sets M, C, R.
4. $Tested := \emptyset$
5. Traverse $\mathcal{S}_r$. For each $[L, z/p]$ in $\mathcal{S}_r$:
   if $\nexists z'.\ [L, z'/p] \in Tested$ then
   $Tested := Tested \cup \{[L, z/p]\}$, and enumerate tests:
   (a) Choose $\langle \bar{l}, \bar{u}\rangle \in [L, z/p]$
   (b) Compute a concrete trace σ from $\langle \bar{l}_0, \bar{0}\rangle$ to $\langle \bar{l}, \bar{u}\rangle$.
   (c) Make Test Cases:
   if $A \in M([L, p])$ then **after** σ **must** A is a relevant test.

   if $a \in C([L, p])$ then **can** $\sigma \cdot a$ is a relevant test.

   if $a \in R([L, p])$ then **after** $\sigma \cdot a$ **must** $\emptyset$ is a relevant test.

□

If a particular point in the symbolic state is of interest, such as an extreme value, this must be computed (step 5a). When a point has been chosen, a trace leading to it from the initial state is computed (step 5b). Finally, in step 5c, a test case can be generated for each of the must, may, and refusal properties holding in that symbolic state, and can finally be output as a test automaton in whatever output format is desired.

It should be noted that the above algorithm generates individual timed Hennessy tests. In general, it is desirable to compose several of these properties into fewer tree structured tests. To facilitate test composition, the traversal and construction of test cases in step 5 should be done differently. A composition algorithm is implemented in RTCAT [32]. Furthermore, the graphs in steps 1 and 2 can be constructed stepwise on a need basis. This could result in a smaller graph and less memory use during its construction because not all equivalence classes may be reachable.

Figure 19 shows some examples of generated test cases from the coffee machine specification in Figure 7a. RTCAT has been configured to select test points in the interior of the equivalence classes.

5 Tool Facilities

We have implemented our approach and algorithms in a prototype tool called RTCAT. RTCAT inputs an ERA specification in AUTOGRAPH format [35]. 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 3.2.

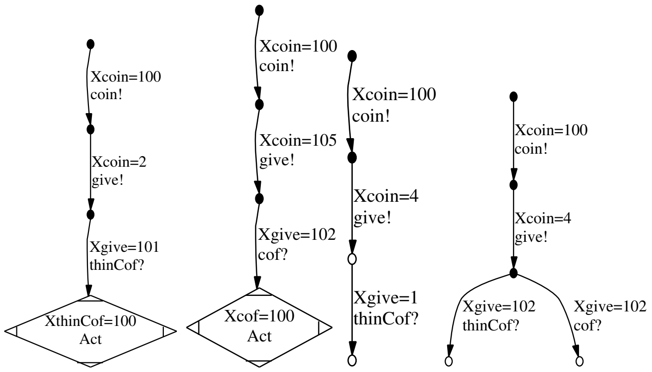


Fig. 19. Example tests generated from the coffee machine in Figure 7. Filled states are **fail** states, and unfilled states are **pass** states. Diamonds contain actions to be refused at the time indicated at the its top. Act is an acronym for all actions.

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 [24] 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.

Test Structure: Tests can be constructed either as individual timed Hennessy tests (Algorithm 18) 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 3.4.

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 [26]. 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 [28]). Its AUTOGRAPH file format parser was reused with some minor modifications to accommodate the ERA syntax. Also its

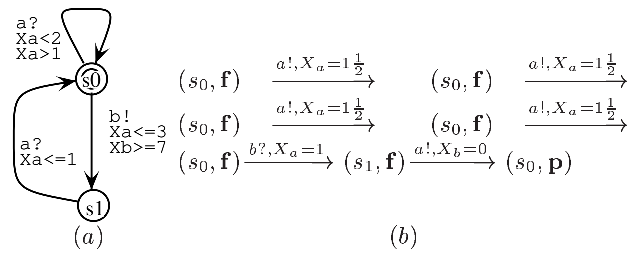


Fig. 20. Simple ERA example

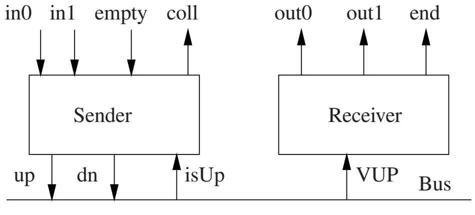


Fig. 21. Overview of the Philips audio protocol.

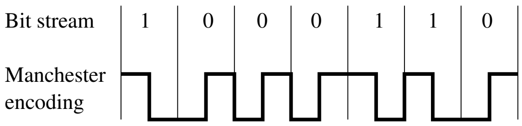


Fig. 22. Manchester encoding of the bit stream 1000110.

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 20a 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 20b; 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 if tests are derived manually.

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 [8]. 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.

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 entities is shown in Figure 21. 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 22, 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 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 23.

Symbol	Value	Meaning			
q	2220	one quarter of a bit slot (220μs)			
d	200	Detection 'just' before `up` (20μs)			
g	220	'Around' 25% and 75% of the bit-slot (22μs)			
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 23. Constants used in the ERA specification of the Philips audio protocol.

The basic operating principle of the sender, shown in Figure 24, 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.

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 receiver, shown in Figure 25, decodes the signal based on occurrences of rising edges. The important states are `L0` and `L1`. The receiver is in `L0` when the last received bit was a zero, and in `L1` when the last bit was a one. According to [6] the model is a direct translation of the decoding algorithm described in the Philips documentation.

The generated tests are exemplified in Figure 26. 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'.

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 (such a detailed description was not available to us) 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 Tool Performance

To analyze the feasibility of our techniques with respect to time and space usage, and also test suite size, we applied RTCAT on a created ERA version of the frequently studied Philips audio protocol [8,6] and a simple token passing protocol. As part of the experiment we measured the number and length of the generated tests, the number of reached (convex) equivalence classes and symbolic states, and the space and time needed to generate the tests and output them to a file.

The platform used in the experiment consists of a Sun Ultra-250 workstation running Solaris 5.7. The machine is equiped with 1 GB RAM and 2x400 MHz CPU's. No extra compiler optimizations were made. The results are tabulated in Table 27.

The size of the produced test suites is in all combinations quite manageable, and constitute test suites that can easily be executed in practice. There is thus a large margin allowing for more test points per equivalence class in the form of several extreme values, generating longer tests by iterating loops in the specification and reachability graph, or by performing experiments for every reached symbolic state as opposed to the selection criterion of equivalence classes only. Moreover, coverage of even larger specifications can also be obtained.

The construction order may influence the number and length of tests because the reached symbolic states are included in the set *Tested* during construction of the reachability graph. Our results show that depth-first construction generates slightly fewer tests than breadth-first, but also considerably longer test suites. This suggests that breadth-first should be used when the most economic covering test suite is desired, and that depth-first should be used when a covering test suite is desired that also checks longer sequences of interactions. Our observation is consistent with most model checking tools which produce shorter counter examples when breadth-first

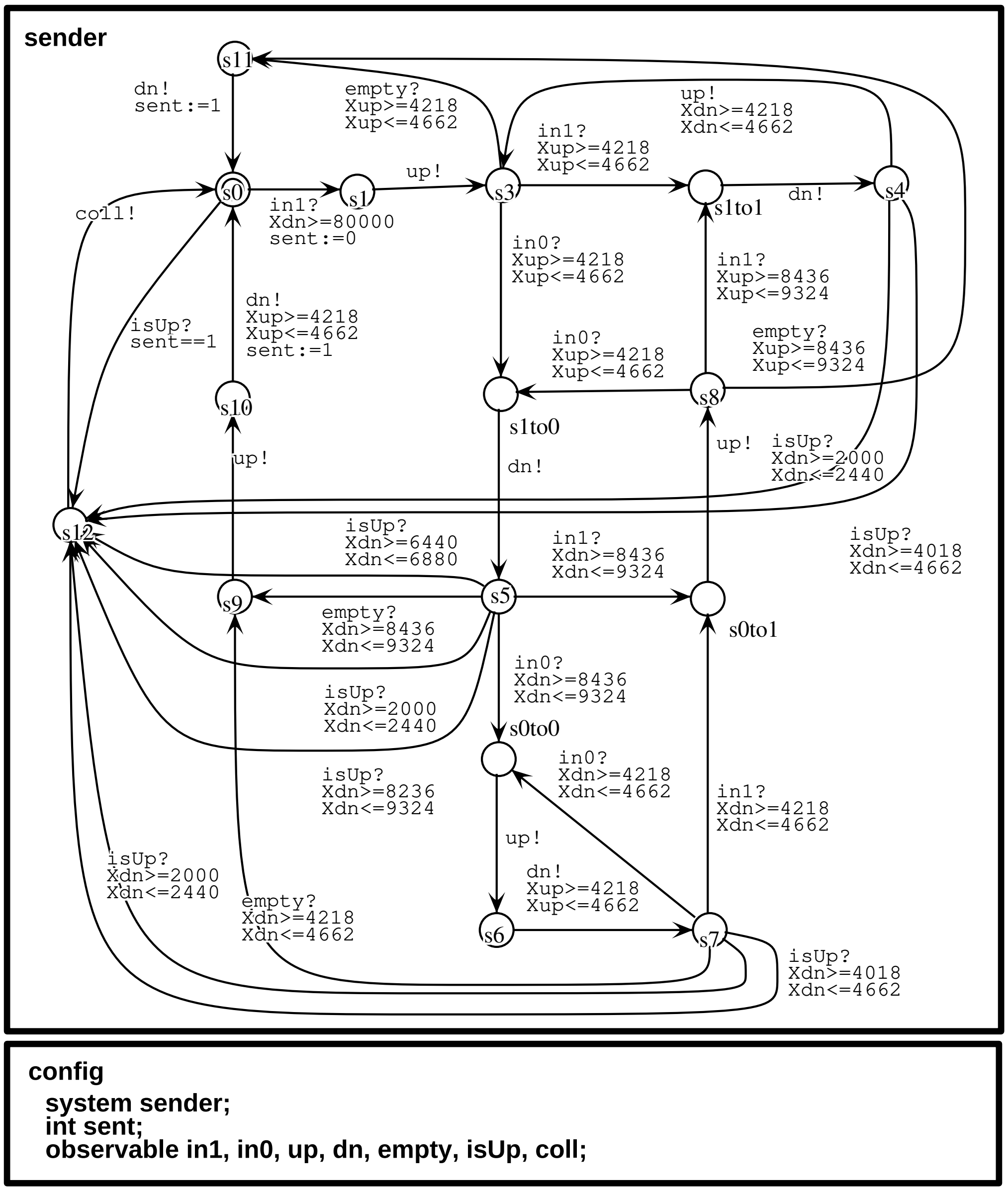


Fig. 24. The sender ERA with collision detection.

traversal is used, but also observe that breadth-first traversal generally consumes more memory during the construction of the state space.

The tabulated figures on the space and time consumption is the maximum observed; generally test composition takes slightly longer and uses a little extra space due to the extra computations and traversals. For the first three specifications, the space and time consumption is quite low, and indicates that fairly large specifications can be handled. However, we have also encountered a problem with our current implementation which occurs for some specifications (such as the token passing protocol), where our application of the symbolic reachability techniques becomes a bottleneck. When the specification uses a large set of active clocks (one per node to measure the token holding time for that node plus one auxiliary in the example), we experience that a large number of symbolic states is constructed in order to terminate the forward reachability analysis. Consequently, an extreme amount of memory is used to guarantee complete coverage. It is important to note that the size of the produced test suite is still quite reasonable. The increased complexity due to clocks that is inherent in the applied symbolic techniques is also experienced in the context of model checking, and here more sophisticated clock reduction algorithms have been applied [20]. These techniques can also be applied to test generation. Running RTCAT on an equivalent one-clock version (only one node may hold the token at a time, and thus one clock suffices) of Tok^7 consumes comparatively little memory.

Also the (non-inherent) way the techniques are implemented in the prototype can be improved. Here reachability analysis

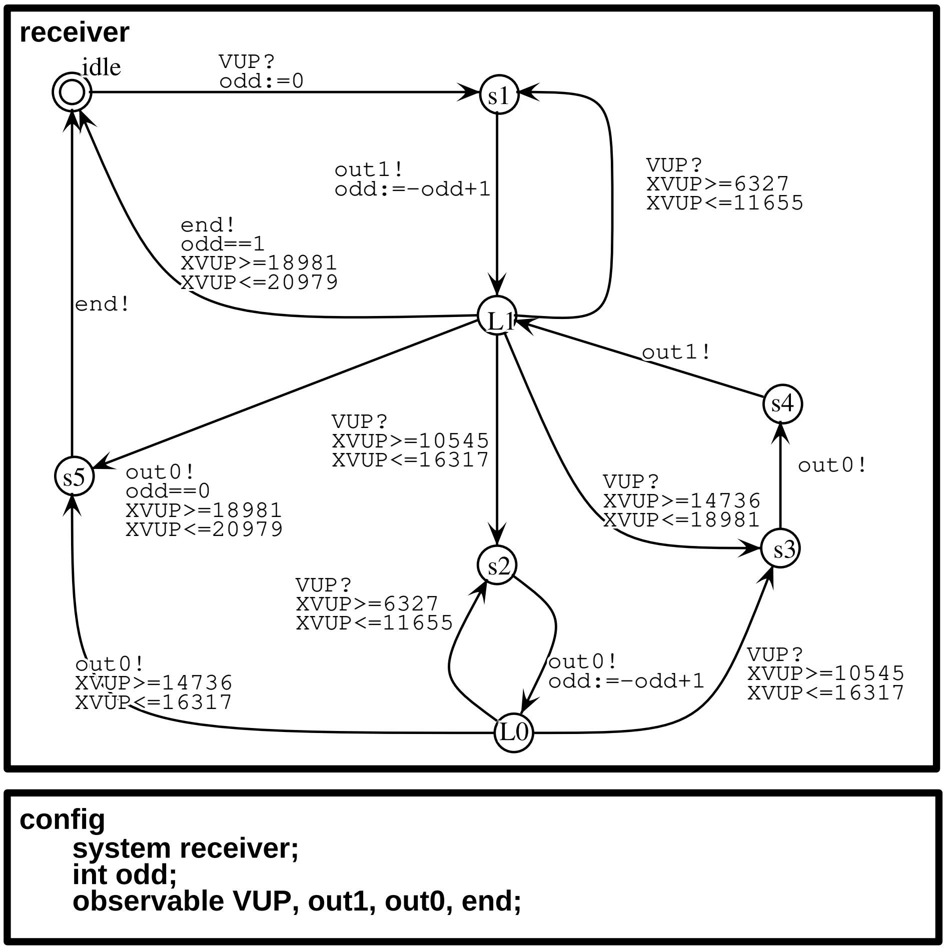


Fig. 25. The receiver ERA.

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.

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

Fig. 26. Examples of tests generated for the Philips Audio Protocol sender.

is performed on the state space that has been partitioned into first equivalence classes, and then, further into convex sets. Although, it simplified implementation of the prototype it results in more and smaller symbolic states than is really necessary from the perspective of reachability analysis. It would therefore be better to apply the reachability analysis on the original specification automaton rather than the partitioned state space, because this should result in larger and fewer symbolic states, with the effect of using less memory. In addition, a new symbolic technique called Clock Difference Diagrams [3] has been developed that more effectively represents non-convex sets of states that zones do. In conclusion, there are several options for reducing the observed problem.

8 Conclusions and Future Work

This paper has presented a new technique for generating real-time tests from a restricted, but determinizable class of timed automata. The underlying testing theory is Hennessy's tests lifted to include timed traces. A principal problem is to generate a sufficiently small test suite that can be executed in

	CofM	Ph-R	Ph-S	Tok[7]
Equivalence Classes	14	60	47	42
Breadth-First				
Symbolic States	17	71	97	15427
Time (s)	1	1	2	541
Memory (MB)	5	5	5	40
C-Number of Tests	16	97	68	71
C-Total Length	45	527	393	574
I-Number of Tests	22	118	85	84
I-Total Length	58	614	467	665
Depth-First				
Symbolic States	17	85	98	7283
Time (s)	1	2	2	158
Memory (MB)	5	5	5	24
C-Number of Tests	16	86	67	60
C-Total Length	45	1619	487	5290
I-Number of Tests	22	118	85	84
I-Total Length	58	2103	587	6321

Table 27. Experimental results from generating tests from the coffee machine (CofM), the Philips audio protocol receiver component (Ph-R), sender component with collision detection (Ph-S), and 7-node token passing protocol (Tok[7]). I=individually generated tests (algorithm 18), C=composed tests.

practice while maintaining a high likelihood of detecting unknown errors and obtaining the desired level of coverage. In our technique, the generated tests are selected on the basis of a coarse equivalence class partitioning of the state space of the specification. We employ the efficient symbolic techniques developed for model checking to synthesize the timed tests, and to guarantee coverage with respect to a coverage criterion which is inspired by classical domain testing criteria. The techniques are implemented in a prototype tool, and the application thereof to a realistic specification shows promising results. The test suite is quite small, and is constructed quickly, and with a reasonable memory usage. Our experiences, however, also indicate a problem with our application of the symbolic reachability analysis, which should be addressed in future implementation work. Compared to previous work based on the region graph technique, our approach appear advantageous.

Much additional work remains to be done. In particular we are examining the possibilities for extending the expressiveness our specification language. It will be important to allow specification and effective test of timing uncertainty, i.e., that an event must be produced or accepted at some (unspecified) point in an interval. Further, it should be possible to specify environment assumptions and to take these into account during test generation. Determinizability is not in general a formal necessity to generate real-time test cases, but one of algorithmic convenience, because testing execution in practice only allows checking of finite behavior. However, further work is becessary to investigate what problems and efficiency can be expected if our techniques are generalized towards a more general timed automaton language. It could also be academically fruitful to develop a formal fault model for our selection criteria, that precisely characterizes the hypothesis required to conclude absolute correctness according to the implementation pre-order. Finally, our techniques should be examined with real applications, and the generated test should be executed against real implementations.

References

1. Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994.
2. Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-Clock Automata: A Determinizable Class of Timed Automata. In *6th Conference on Computer Aided Verification*, 1994. Also in LNCS 818.
3. Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Computer Aided Verification (CAV'99)*, volume LNCS 1633, pages 22–24. Springer Verlag, July 1999. Trento, Italy.
4. Boris Beizer. *Software Testing Techniques*. International Thompson Computer Press, 1990. 2nd edition, ISBN 1850328803.
5. Richard Bellman. Dynamic programming, 1957.
6. Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang Yi. Verification of an Audio Protocol with Bus Collision using UppAal. In *9th Intl. Conference on Computer Aided Verification*, pages 244–256, 1996. LNCS 1102.
7. Bernhard Steffen and Rance Cleaveland, editor. *International Journal on Software Tools for Technology Transfer (STTT)*, volume 1/1+2. Springer, December 1997.
8. Doeko Bosscher, Indra Polak, and Frits Vaandrager. Verification of an Audio Protocol. TR CS-R9445, CWI, Amsterdam, The Netherlands, 1994. Also in LNCS 863, 1994.
9. Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly Symbolic Model-Checking for Real-Time Systems. In *1997 IEEE Real-Time Systems Symposium, RTSS'97*, San Fransisco, USA, December 1996. IEEE Computer Society Press.
10. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.
11. Marius Bozga, Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Protocol verification with the ALDBARAN toolset. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1+2):166–184, 1997.
12. V. Braberman, M. Felder, and M. Marré. Testing Timing Behaviors of Real Time Software. In *Quality Week 1997. San Francisco, USA.*, pages 143–155, April-May 1997 1997.
13. Ed Brinksma. A Theory for the Derivation of Tests. In *Protocol Specification Testing and Verification VIII (PSTV'88)*, pages 63–74, 1988.
14. Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
15. Rachel Cardell-Oliver and Tim 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.

16. R. Castanet, Ousmane Koné, and Patrice 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.
17. Duncan Clarke and Insup Lee. Testing Real-Time Constraints in a Process Algebraic Setting. In *17th International Conference on Software Engineering*, 1995.
18. Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.
19. Rance Cleaveland and Matthew Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.
20. Conrado Daws and Sergio Yovine. Reducing the Number of Clock Variables of Timed Automata. In *1996 IEEE Real-Time Systems Symposium, RTSS'96*, Washington, DC, USA, december 1996. IEEE Computer Society Press.
21. Ren G. de Vries and Jan Tretmans. On-the-fly conformance testing using SPIN. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):382–393, 2000.
22. David 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.
23. Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat 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.
24. Gerard J. Holzmann. *Design and Validation of Computer Protocols*, chapter 11. Prentice Hall, New Jersey, U.S.A, 1991. ISBN 0-13-539925-4.
25. Alain Kerbrat, Thierry Jéron, and Roland Groz. Automated Test Generation from SDL Specifications. In *Ninth SDL Forum*, 21-25 June 1999. Montral, Qubec, Canada.
26. Eleftherios Koustsofios and Stepen 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.
27. Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang 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.
28. Kim G. Larsen, Paul Pettersson, and Wang Yi. UppAal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.
29. Philippe Leblanc. OMT and SDL based techniques and tools for design, simulation and test production of distributed systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1+2):153–165, 1997.
30. Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.
31. R. De Nicola and M.C.B Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.
32. Brian Nielsen. *Specification and Test of Real-Time Systems*. PhD thesis, Department of Computer Science, Aalborg University, Denmark, april 2000.
33. Jan Peleska and Bettina 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.
34. Jan Peleska and Cornelia Zahlten. Test Automation for Avionic Systems and Space Technology. In *GI Working Group on Test, Analysis and Verification of Software*, 1999. Munich, Extended Abstract.
35. Annie Ressouche, Robert de Simone, Amar Bouali, and Valérie Roy. The FCTOOLS User Manual. Technical Report `ftp://ftp-sop.inria.fr/meije/verif/fc2.userman.ps`, INRIA Sophia Antipolis.
36. John Rushby. Formal Methods: Instruments of Justification or Tools of Discovery. In *Nordic Seminar on Dependable Computing Systems (NSDCS'94)*, Lyngby, Denmark, August 1994. Tutorial.
37. Holger Schlingloff, Oliver Meyer, and Thomas Hülsing. Correctness Analysis of an Embedded Controller. In *Data Systems in Aerospace (DASIA99). ESA SP-447, Lisbon, Portugal*, pages 317–325, 1999.
38. Michael Schmitt, Beat Koch, Jens Grabowski, and Dieter Hogrefe. Autolink - Putting formal test methods into practice. In *11th Int. Workshop on Testing of Communicating Systems (IWTCS'98)*, Tomsk, Russia, September 1998.
39. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. TR CTIT 97-17, University of Twente, 1997. To appear in *Theoretical Computer Science*.
40. Volker Diekert, Paul Gastin, Antoine Petit. Removing epsilon-Transitions in Timed Automata. In *14th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1997*, pages 583–594, Lübeck, Germany, February 27 - March 1 1997. Lecture Notes in Computer Science, Vol. 1200, Springer, 1997, ISBN 3-540-62616-6.
41. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint Solving. In *7th Int. Conf. on Formal Description Techniques*, pages 223–238, 1994. North-Holland.