Automated Test Generation from Timed Automata

Brian Nielsen and Arne Skou

Aalborg University
Department of Computer Science
Fredrik Bajersvej 7E
DK-9220 Aalborg, Denmark
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.

1 Background

Testing consists of executing a program or a physical system with the intention of finding undiscovered errors. In typical industrial projects, as much as a third of the total development time is spent on testing, and it therefore constitutes a significant portion of the cost of the product. Since 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 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 [7, 20, 23, 26], and commercial test generations tools are emerging [17, 24]. Typically, a test generation tool inputs some kind of finite state machine description of the behavior required of the implementation. A formalized *implementation relation* describes exactly what it means for an implementation to be correct with respect to a specification. The tool interprets the specification or transforms it to a data structure appropriate for test generation, and then computes a set of test sequences. Since exhaustive testing is generally infeasible, it must select only a subset of tests for execution. Test selection can be based on manually stated test purposes, or on a coverage criterion of the specification or implementation.

However, these 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. 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. Since real-time systems are often safety critical, the time dimension must be tested thoroughly and systematically. 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 [21] for concurrent systems, which we have generalized to take time into account. We propose to select test cases by partitioning the state space into coarse grained equivalence classes which 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 [3]. We regard 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 [15, 6, 28, 4, 18].

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 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, and the

symbolic reachability methods. Section 4 presents the test generation algorithm. Section 5 contains our experimental results. Section 6 concludes the paper and suggests future work.

2 Related Work

Springintveld et al. proved in [27] 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 [16]. Although the required discretization step size ($1/(|X|+2)$, where $|X|$ is the number of clocks) in [16] is more reasonable than [27], it still appears to be too small for most practical applications because too many tests are generated. Both of these techniques are based on 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 [9] how to derive checking sequence from a *discrete time*, *deterministic*, timed transition system model. Their approach is implemented in a tool which 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.

Clarke and Lee [11, 12] 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., since their constraint graphs must be acyclic this only permits specification of finite behaviors. Their domains are “nice” linear intervals which 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. [8] 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 [10] 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 which 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 [20].

3 Preliminaries

3.1 Hennessy Tests

In Hennessy's testing theory [21] 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. So, 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 $\mathcal{L}_{\mathrm{tlts}}$: (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 is this is not the case (i.e. σ is not a prefix).

Definition 1. *The Testing Preorder* $\sqsubseteq_{\mathbf{te}}$:

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

□

Specifications and implementations are compared by the tests they pass. 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 1 requires satisfaction on both the must and may preorders.

A must test $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ 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 [13]) 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 $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ (or $\mathbf{after}\ \sigma\ \mathbf{must}\ \emptyset$), 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 $\mathbf{can}\ \sigma\ (\mathbf{after}\ \sigma\ \mathbf{must}\ \emptyset)$ 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. 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.

Definition 2. *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. 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.
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]. 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.

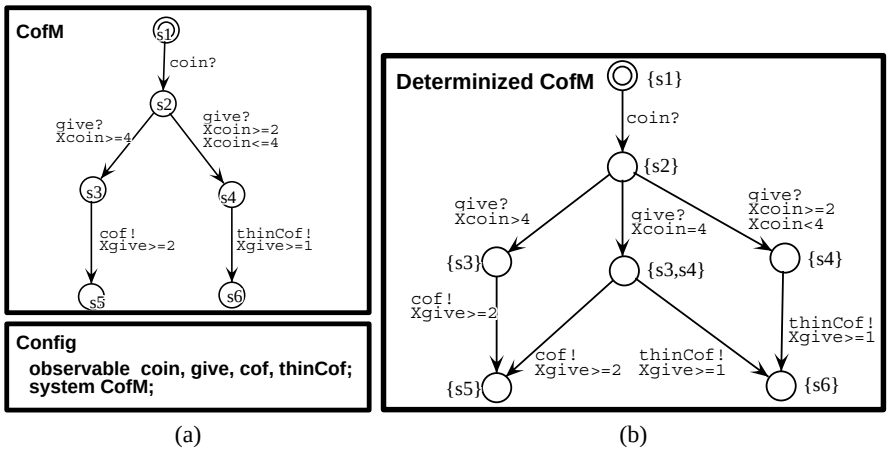

Fig. 3. *ERA specification of a coffee vending machine (a), and determinized machine (b).*

Figure 3a 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 his eager 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 3b 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.

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 [15, 6, 28, 4, 18]. Specifically, we shall employ similar techniques as those developed for the UPPAAL tool [28, 4, 18].

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. A symbolic state $[\bar{l}, z]$ represents a (infinite) set of states: $\{\langle \bar{l}, \bar{u} \rangle \mid \bar{u} \in z\}$. Forward reachability analysis starts in the initial state, and computes the symbolic states that can be reached by executing an action or a delay from an existing one. 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. A concrete timed trace to a given state or set of states 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.

Zones can be represented and manipulated efficiently by the *difference bound matrix* (DBM) data structure. DBMs were first applied to represent clock differences by Dill in [15]. A DBM represents clock difference constraints of the form $x_i - x_j \prec c_{ij}$ by a $(n+1) \times (n+1)$ matrix such that c_{ij} equals matrix element (i, j), where n is the number of clocks, and $\prec \in \{\leq, <\}$.

4 A Test Generation Algorithm

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

4.1 State 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. 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.

We propose a 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 [22]. 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), $\mathbf{after}\ a\ \mathbf{must}\ \emptyset$ (a *refusal* property), and $\mathbf{can}\ a$ (a *may* property) are made at least once in each class.

From each control location L (recall that a location in a deterministic 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 contiguous convex equivalence classes, 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 4.

Definition 4. *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 2 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. $\Psi_{\mathrm{dnf}}(L) = \bigcup_{P_E \in \Psi(L)} \mathrm{GDNF}(P_E)$

□

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 2. It also 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 since 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.

Each equivalence class $[L, p]$ can now be decorated with the action sets M, C, R defined in Definition 5.

Definition 5. *Decorated Equivalence Classes*:
Define $\mathrm{Must}([L,p]) = \{A \mid \exists \langle L, \bar{u}\rangle \in [L,p].\ \langle L, \bar{u}\rangle \models \mathbf{after}\ \epsilon\ \mathbf{must}\ A\}$
$\mathrm{Sort}([L,p]) = \{a \mid \exists \langle L, \bar{u}\rangle \in [L,p].\ \langle L, \bar{u}\rangle \xrightarrow{a}\}$

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

□

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 $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ is a test to be passed for that class. So is $\mathbf{after}\ \sigma \cdot a\ \mathbf{must}\ \emptyset$ if $a \in R([L,p])$, and $\mathbf{can}\ \sigma \cdot a$ 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. The must sets can be reduced to $M([L,p]) = min_{\subseteq}\mathrm{Must}[L,p]$. The actions observed during the execution of a must test can be removed from the may tests, i.e., $C([L,p]) = \mathrm{Sort}([L,p]) - \bigcup_{A \in M([L,p])} A$.

4.2 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 are 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 4 (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 6. This definition can easily be turned into an algorithm for constructing the equivalence class graph.

Definition 6. *Equivalence Class Graph*:
The nodes and edges are defined inductively as:

1. The set $\{[L_0,p] \mid L_0 = \{\bar{l}_0\},\ \ p \in \Psi_{\mathrm{dnf}}(L_0),\ \text{and}\ p \neq \emptyset\}$ 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. 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 not lost, 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. The equivalence class graph for the coffee machine is depicted in Figure 7.

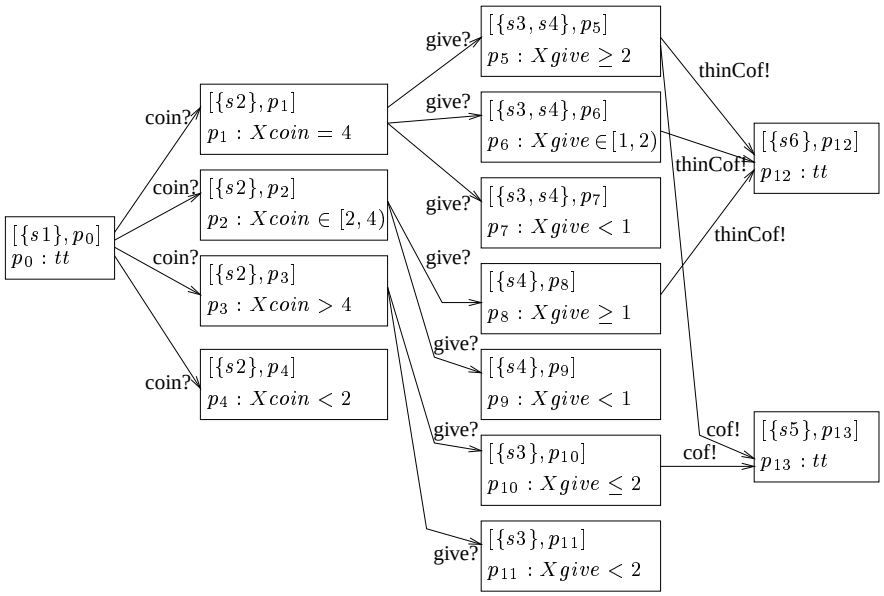

Fig. 7. *Equivalence class graph for the coffee machine.*

4.3 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. Moreover, the use of reachability analysis gives a termination criterion for this 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 8 presents the main steps of our generation procedure. Step 1 constructs the equivalence class graph as described in Section 4.1. 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 5 in step 3. The boolean flag $toBeTested$ indicates whether test cases should be made for this symbolic state or they should be omitted. If no tests should be made, the only actions executed from this state will be those necessary to reach other symbolic states. Normally this flag would be set only the first time an equivalence class is reached during the forward reachability analysis in the previous step. Subsequent passes over the same class would hence be ignored. This ensures that each simple deadlock property is only generated once per equivalence class, and thus reduces the number of produced test cases. Different settings of this flag permit other strategies to be easily implemented. Other strategies could be to test all reached symbolic states, or only test certain designated locations deemed critical by the user.

Algorithm 8. *Overall Test Case Generation Algorithm*:
input: ERA specification $\mathcal{S}$.
output: A complete covering set of timed Hennessy tests to be passed.

1. Compute $\mathcal{S}_p$ = Equivalence Class Graph($\mathcal{S}$).
2. Compute $\mathcal{S}_r$ = Reachability Graph($\mathcal{S}_p$).
3. Label every $[L, z/p] \in \mathcal{S}_r$ with the sets M, C, R, and boolean flag $toBeTested$.
4. Traverse $\mathcal{S}_r$. For each $[L, z/p]$ in $\mathcal{S}_r$:
   if $toBeTested([L, z/p])$ then enumerate tests:
   (a) Choose $\langle \bar{l}, \bar{u} \rangle \in [L, z/p]$
   (b) Compute a concrete timed trace $\sigma \in \mathcal{S}_r$ from $\langle \bar{l}_0, \bar{0} \rangle$ to $\langle \bar{l}, \bar{u} \rangle$.
   (c) Make test cases to be passed:
   if $A \in M([L, p])$ then $\mathbf{after}\ \sigma\ \mathbf{must}\ A$ is a test.
   if $a \in C([L, p])$ then $\mathbf{can}\ \sigma \cdot a$ is a test.
   if $a \in R([L, p])$ then $\mathbf{after}\ \sigma \cdot a\ \mathbf{must}\ \emptyset$ is a test.

□

Step 4 contains the generation process itself. If a particular point in the symbolic state is of interest, such as an extreme value, this must be computed (step 4a). When a point has been chosen, a trace leading to it from the initial state is computed (step 4b). Finally, in step 4c, 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 4 should be done differently. A composition algorithm is implemented in RTCAT. Furthermore, the graphs in steps 1 and 2 can be constructed on-the-fly. Since not all equivalence classes may be reachable, this could result in a smaller graph and less memory use during its construction.

5 Experimental Results

RTCAT accepts ERA specifications in AUTOGRAPH format [25]. 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. Other features are described in [22]. 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 [19]). 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.

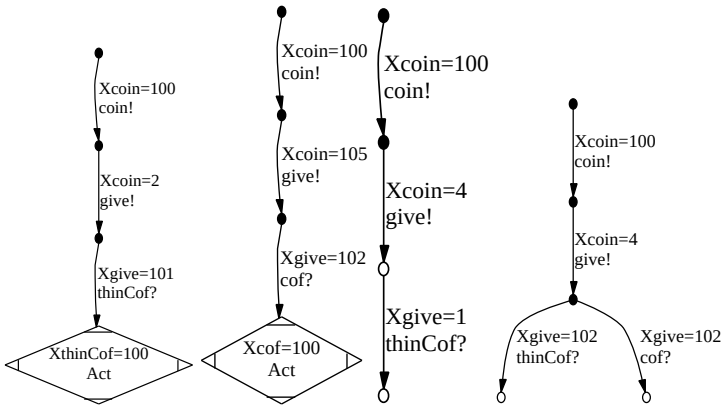

Fig. 9. *Example tests generated from the coffee machine in Figure 3. Filled states are* **fail** *states, and unfilled states are* **pass** *states. Diamonds contain actions to be refused at the time indicated at the its top.* Act *is an acronym for all actions.*

Figure 9 shows some examples of generated test cases from the coffee machine specification in Figure 3a. RTCAT has been configured to select test points in the interior of the equivalence classes. To analyze the feasibility of our techniques we have created an ERA version of the frequently studied Philips audio protocol [5, 4] and a simple token passing protocol, applied RTCAT, and 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 ERA models can be found in [22]. 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 was done to the code. The results are tabulated in Table 10.

The size of the produced test suites is in all combinations quite manageable, and constitute test suites that could easily be executed in practice. There is thus a large margin allowing for more test points per equivalence class, or longer tests. Moreover, coverage of even larger specifications can also be obtained. Since the reached sym-

bolic states are labeled $toBeTested$ during construction of the reachability graph, the construction order may influence the number and length of tests. 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.

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

Table 10. *Experimental results from generating tests from the coffee machine, the Philips audio protocol receiver component, sender component with collision detection, and 7-node token passing protocol. I=individually generated tests (algorithm 8), C=composed tests.*

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. 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. We believe that this problem can be alleviated by applying the reachability analysis on the original specification automaton rather than as presently done on the equivalence class graph. This should result in larger and fewer symbolic states. Further, more sophisticated clock reduction algorithms could be applied [14], e.g., in the token passing protocol only one node may hold the token at a time, and thus one clock suffices.

6 Conclusions and Future Work

This paper 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 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. The techniques are implemented in a prototype tool. 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 other work remain to be done. In particular we are examining the possibilities for generalizing 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. 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] Boris Beizer. *Software Testing Techniques*. International Thompson Computer Press, 1990. 2nd edition, ISBN 1850328803.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Duncan Clarke and Insup Lee. Testing Real-Time Constraints in a Process Algebraic Setting. In *17th International Conference on Software Engineering*, 1995.

[12] 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.

[13] Rance Cleaveland and Matthew Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[14] 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.

[15] 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.

[16] 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.

[17] Alain Kerbrat, Thierry Jéron, and Roland Groz. Automated Test Generation from SDL Specifications. In *Ninth SDL Forum*, 21-25 June 1999. Montral, Qubec, Canada.

[18] 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.

[19] 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.

[20] 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.

[21] R. De Nicola and M.C.B Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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*.

[28] 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.