AN ABSTRACTION TECHNIQUE FOR REAL-TIME VERIFICATION

Edmund M. Clarke, Flavio Lerda, Muralidhar Talupur
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
{flerda,tmurali,emc}@cs.cmu.edu

Abstract In real-time systems, correctness depends on the time at which events occur. Examples of real-time systems include timed protocols and many embedded system controllers. Timed automata are an extension of finite-state automata that include real-valued *clock variables* used to measure time. Given a timed automaton, an equivalent finite-state region automaton can be constructed, which guarantees decidability. Timed model checking tools like UPPAL, KRONOS, and RED use specialized data structures to represent the real-valued clock variables. A different approach, called integer-discretization, is to define clock variables that can assume only integer values, but, in general, this does not preserve continuous-time semantics.

This paper describes an implicit representation of the region automaton to which ordinary model checking tools can be applied directly. This approach differs from integer discretization because it is able to handle real-valued clock variables using a finite representation and preserves the continuous-time semantics of timed automata. In this framework, we introduce the GOABSTRACTION, a technique to reduce the size of the state space. Based on a conservative approximation of the region automaton, GOABSTRACTION makes it possible to verify larger systems. In order to make the abstraction precise enough to prove meaningful properties, we introduce auxiliary variables, called `Go` variables, that limit the drifting of clock variables in the abstract system. The paper includes preliminary experimental results showing the effectiveness of our technique using both symbolic and bounded model checking tools.

Keywords: Abstraction; model checking; real-time systems; timed automata.

Overview

Real-time systems are a class of systems whose correctness depends on the time at which events occur. Examples include embedded controllers,

time triggered systems, and timed protocols. In fact, most safety critical systems are real-time systems as they require guarantees on the timing of events. For instance, in order for the braking system of a car to be correct, it is not sufficient for the correct output to be produced, but it has to be produced within a given time bound.

Model checking is widely used in the semiconductor industry and it has been successful for software. For example, the model checker SLAM [3] is the basis for the *Driver Verifier*, which is currently being distributed by Microsoft as part of their Device Driver Software Development Kit. Moreover, all device drivers must be verified using the Driver Verifier in order to be certified by Microsoft's Windows Hardware Quality Labs. However, finite-state model checking cannot be applied directly to real-time systems because time is modeled as a continuous, real-valued quantity. The success of finite-state model checking has spurred the development of model checking techniques for infinite-state systems, including real-time systems.

Real-time systems are often modeled using *timed automata* [2]. Timed automata are an extension of finite-state automata that include a set of *clock variables* to keep track of time. The transitions of a timed automaton are labeled with *clock constraints* that must hold when a transition is taken, and sets of *clock variables to be reset* after a transition occurs. To specify properties of timed automata, extensions of ordinary temporal logics, e.g., *Timed Computation Tree Logic* (TCTL) [1], have been proposed.

In recent years, there has been extensive work on verification of real-time systems. Alur et al.[1, 2] developed the theoretical foundations for much of the work in this area. They introduced *timed automata*, an extension of finite-state automata that can be used to model real-time systems. They proposed the *region graph construction*, which maps questions about an (infinite-state) real-time system into questions about a corresponding finite-state automaton. Many tools and techniques for real-time verification are based on this work but employ specialized data structures to represent clock variables: Difference Bounded Matrices [7], Region Encoding Diagrams [17], Clock-Restriction Diagrams [20], and Difference Decision Diagrams [13], just to name a few. Henzinger and Kupferman [9] showed how to reduce the problem of checking a timed temporal logic property of a timed automaton to checking an (untimed) temporal logic property of a property of the region automaton. Therefore, from now on, we will only consider untimed temporal logic properties. Other approaches have also been proposed. For instance, extensive work has been done on integer discretization based techniques [5, 10, 4, 11], where real-valued clock variables are replaced by integer-valued vari-

ables. While these techniques in general do not preserve the continuous-time semantics of timed-automata, they are sound for a class of real-time systems and properties [10].

In this paper, we explore techniques for the verification of timed automata that are based on the region graph construction but do not use specialized data structures to represent clock variables. Preliminary work in this area is due to Gōllü et al. [8], however, no implementation or experimental results were presented by the original authors. These techniques are usually referred to as *discretization* techniques, but they are radically different from the integer-discretization techniques mentioned above. The main difference is that the former provide a finite (discrete) representation for sets of real-valued clock variables while the latter is able to handle only integer-valued clock variables. In this paper, we describe a new implicit representation of the region automaton. The representation is implicit in that we do not enumerate regions or transitions explicitly. The resulting system can be verified using existing model checking tools. Our representation of clock regions is similar to the one of Wang et al. [18], however, their approach is based on a specialized data structure for symbolic model checking and it cannot be used with other model checking tools.

The region graph construction [1] is a well known technique for model checking timed automata. A state of a timed automaton is defined as a pair made of a location and a valuation of the clock variables. A clock region is a (possibly infinite) set of clock valuations. The region graph construction defines a bisimulation between the states of a timed automaton and a finite set of clock regions. The result of the construction is a region automaton, a finite-state automaton that is bisimilar to the original timed automaton. The bisimulation defined in [1] preserves temporal logic properties. Verification of a property of a timed automaton is reduced to the verification of the same property on the corresponding region automaton. The region automaton is, by construction, finite, therefore, ordinary model checking techniques can be applied to it. However, since the region automaton can be exponential in the size of the original timed automaton, existing tools like Uppaal [12], Kronos [21] and Red [19] treat clock variables differently from discrete state variables and use specialized data structures to represent clock regions.

Since the region automaton is, in the worst case, exponential in the number of clock variables [2], we introduce a new abstraction technique called GoAbstraction that addresses this blow up. Approaches that use a representation similar to ours, e.g. [8, 18], do not have a similar abstraction technique.

Abstractions have been widely used in hardware and software model checking to improve the performance of verification. Predicate abstraction has been applied to timed automata by Möller et al. [14] and Sorea [15]. This approach is based on identifying a set of predicates that is sufficient to discriminate between any two clock regions and uses abstraction/refinement to find a minimal subset of these predicates that is sufficient to perform the verification. Tripakis and Yovine [16] define an abstraction that removes the actual value of the delays to obtain a timeless system, which is finite-state. Our approach, instead, is based on merging clock regions that differ only in the ordering of fractional parts.

In the region graph construction, a clock region corresponds to a set of clock valuations that are equivalent according to the bisimulation relation presented in [1]. One of the conditions necessary for two clock valuations v and v' to be equivalent is that the *ordering of the fractional parts* of each pair of clock variables is the same in both valuations. For instance, if the fractional part of clock c_1 is less than the fractional part of clock c_2 in v, the same must hold for v', even if the actual values may differ. This is necessary to precisely compute the successors of a given clock region. However, given n clock variables, there are $n!$ possible orders of their fractional parts, and, in principle, $n!$ different clock regions. This can cause an exponential blow up in the number of states in the region automaton, which can lead to intractability.

Our approach abstracts the relative ordering between the fractional parts. By doing so, we obtain an over-approximation of the behavior of the system, where precise information is lost. Regions that differ only because of the ordering of the fractional parts of some clock variables are merged into a single abstract clock region. By reducing the number of clock regions, we decrease the number of states in the region automaton and, therefore, we obtain a smaller state space. However, while the abstraction is safe and it is guaranteed to preserve the validity of properties, it may introduce spurious counterexamples.

The abstraction scheme as presented so far is too coarse. The problem is that, as we discard the relative ordering between clock variables, we allow them to drift apart unboundedly. In order to make the abstraction more precise, we introduce auxiliary variables, called `Go` variables, that keep track of the way clock variables evolve and limit the drifting to at most one time unit.

In our preliminary experiments, we show how GoAbstraction is sufficient to prove properties for a real-time protocol, namely Fischer's mutual exclusion protocol, that could not be established with a naive abstraction scheme that did not make use of the `Go` variables.

The reminder of the paper is organized as follows. Section 1 recalls some useful definitions. Our discretization is presented in Section 2, and GoAbstraction is introduced in Section 3. Section 4 contains some preliminary experimental results and Section 5 gives conclusions and directions for future work.

1. Preliminaries

1.1 Timed Automata

Timed automata are a formalism used to model real-time systems. They are an extension of finite-state automata that include a set of real-valued *clock variables* used to measure time. Transitions of a timed automaton are labeled with a *clock constraint* and a set of clock variables known as the *reset set*. A transition can be taken only if the clock constraint associated with it is true in the current state. After a transition is taken, the values of the clock variables in the reset set are set to zero.

Definition 1 (Clock Constraints) *A* clock constraint *is a Boolean combination of equalities and inequalities involving a single clock variable x and an integer constant c (i.e., $x < c$, $x \leq c$, $x = c$, $x \geq c$, and $x > c$).*

The set of all possible clock constraints over a set of clock variables X is denoted by $C(X)$.

Definition 2 (Clock Valuations) *A* clock valuation *over a set of clock variables X is a function $v : X \to \mathbb{R}^+$ that assigns to every clock variable in X a non-negative real value.*

The set of all possible clock valuations over a set of clock variables X is denoted by $V(X)$. Let v_0, called the *zero clock valuation*, be the clock valuation that assigns the value zero to all clock variables. Given a clock valuation $v \in V(X)$ and a non-negative real value $\delta \in \mathbb{R}^+$, we denote by $v + \delta \in V(X)$ the clock valuation that maps every clock variable $x \in X$ to the value $v(x) + \delta$. Given a clock valuation $v \in V(X)$ and a reset set $\lambda \subseteq X$, we denote by $v[\lambda = 0] \in V(X)$ the clock valuation that maps every clock variable x in λ to zero and every clock variable x not in λ to the same value v does. Given a clock constraint $g \in C(X)$ and a valuation $v \in V(X)$, v satisfies g if and only if the expression obtained by replacing in g every occurrence of a clock variable x with the value $v(x)$ evaluates to true.

Definition 3 (Timed Automaton) *A* timed automaton *is a 5-tuple $A = (Q, X, q_0, I, T)$ where Q is a finite set of locations; X is a finite set of*

real-valued clock variables; $q_0 \in Q$ is an initial location; $I : Q \to 2^{V(X)}$ is a location invariant, a function that assigns to every location a set of valid valuations; and $T \subseteq Q \times C(X) \times 2^X \times Q$ is a set of discrete transitions, such that $(q, g, \lambda, q') \in T$ if and only if there is a discrete transition from location q to location q' labeled with the clock constraint g and the reset set λ.

The state of a timed automaton A is a pair (q, v) such that $q \in Q$ is a location and $v \in V(X)$ is a clock valuation. Timed automata allow two types of transitions: *(i)* time transitions, which correspond to the passing of time; and *(ii)* discrete transitions, which correspond to the discrete transitions of the automaton. A *time transition* is labeled by a positive real value δ and maps state (q, v) into state $(q, v + \delta)$ if for all non-negative real values $\delta' \leq \delta$, $v + \delta'$ belongs to the invariant $I(q)$. A *discrete transition* is labeled by $(q, g, \lambda, q') \in T$ and maps state (q, v) into state $(q', v[\lambda = 0])$ if v satisfies the clock constraint g and $v[\lambda = 0]$ belongs to the invariant $I(q')$.

1.2 Region Graph Construction

A state of a timed automaton is a pair made of a location and a clock valuation. Therefore, the set of possible states is infinite, as the clock variables are assigned values from $\mathbb{R}^+$. Model checking was developed as a technique for automatically verifying properties of finite-state systems. As such, it is not directly applicable to timed automata, since they may have an infinite number of states.

Alur et al. [1] proposed the *region graph construction* as a way to make verification of real-time systems feasible. Given a timed automaton, the region graph construction produces a *region automaton*, a finite-state automaton that is bisimilar to the original timed automaton. Model checking can then be performed on the region automaton, which satisfies the same set of properties as the original timed automaton.

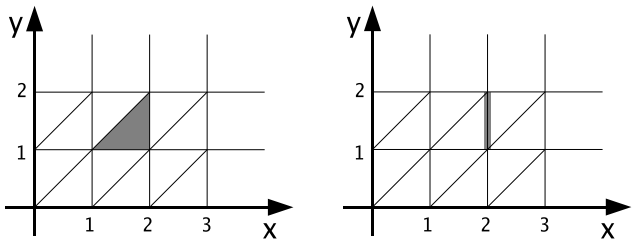
Given a timed automaton A, for each clock variable $x \in X$, let M_x be the largest constant against which x is compared in the clock constraints associated with the discrete transitions of A. We call M_x the *maximum constant value* of clock variable x in the timed automaton A. Let $\lfloor x \rfloor$ be the integer part of clock variable x, and $\langle x \rangle = x - \lfloor x \rfloor$ be its fractional part.

Definition 4 (Equivalent Clock Valuations) *Given a set of clock variables X and their maximum constant values M_x, two clock valuations $v_1, v_2 \in V(X)$ are* equivalent*, $v_1 \approx v_2$, if and only if:*

- *For all $x \in X$, either $\lfloor v_1(x) \rfloor = \lfloor v_2(x) \rfloor$ or both $v_1(x)$ and $v_2(x)$ are greater than M_x;*
- *For all $x \in X$ such that $v_1(x) \leq M_x$, $\langle v_1(x) \rangle = 0$ if and only if $\langle v_2(x) \rangle = 0$; and*
- *For all $x, y \in X$ such that $v_1(x) \leq M_x$ and $v_1(y) \leq M_y$, $\langle v_1(x) \rangle \lhd \langle v_1(y) \rangle$ if and only if $\langle v_2(x) \rangle \lhd \langle v_2(y) \rangle$, for $\lhd \in \{<, =, >\}$.*

As an example, consider Figure 1. Each point on one of the two diagrams corresponds to a clock valuation, each shaded area to a set of equivalent clock valuations. The shaded area on the left shows the clock valuations such that $\lfloor x \rfloor$ and $\lfloor y \rfloor$ are equal to 1 and $\langle y \rangle$ is smaller than $\langle x \rangle$. The shaded area on the right represents the clock valuations such that $\lfloor x \rfloor = 2$, $\lfloor y \rfloor = 1$, $\langle y \rangle < \langle x \rangle$, and $\langle x \rangle = 0$.

Figure 1. The shaded area in each diagram represents a set of equivalent clock valuations.



The first two conditions of Def. 4 guarantee that given two equivalent clock valuations, they satisfy the same set of clock constraints. Given a clock constraint $x \lhd c$, the validity of the constraint can be decided by knowing the integer part of x and whether the fractional part of x is equal to zero.

The third condition is needed to guarantee that, given two equivalent clock valuations, as time passes, they will reach clock valuations that are equivalent. Consider again Figure 1. For all clock valuations represented by the shaded area in the diagram on the left $\langle y \rangle < \langle x \rangle$. As a consequence, as time passes, since both variables are incremented at the same rate, clock variable x will reach the value 2 before clock variable y does. Therefore, the set of clock valuations such that $\lfloor x \rfloor = 2 \wedge \lfloor y \rfloor = 1 \wedge \langle x \rangle < \langle y \rangle \wedge \langle x \rangle = 0$ is reachable. The shaded area in the diagram on the right represents this set of clock valuations. If we did not know the ordering of the fractional parts of x and y, two other sets of equivalent clock valuations would also be reachable, $\lfloor x \rfloor = \lfloor y \rfloor = 2 \wedge \langle x \rangle = \langle y \rangle = 0$ and $\lfloor x \rfloor = 1 \wedge \lfloor y \rfloor = 2 \wedge 0 = \langle y \rangle < \langle x \rangle$.

DEFINITION 5 (CLOCK REGION) *A* clock region μ *is an equivalence class of the relation* $\approx$ *defined above.*

Let the set of clock regions of the automaton A be denoted by $\Gamma(A)$. $\Gamma(A)$ is finite by construction. Since all valuations in a clock region satisfy the same set of clock constraints, a region μ satisfies a clock constraint c if and only if every clock valuation $v \in \mu$ satisfies c. Given a clock region μ, we define $\mu' = \mu[\lambda = 0]$ to be the clock region such that, for all clock valuations $v \in \mu$, $v[\lambda = 0]$ belongs to μ'.

DEFINITION 6 (TIME SUCCESSOR) *Given a clock region* μ*, a clock region* $\mu' \neq \mu$ *is a* time successor *of* μ *if and only if there exists a clock valuation* $v \in \mu$ *and a positive real value* δ*, such that* $v + \delta \in \mu'$ *and for all non-negative real values* $\delta' < \delta$*,* $v + \delta'$ *belongs either to* μ *or* μ'*.*

Notice that each clock region μ has at most one time successor because of the way we defined the equivalence relation $\approx$ on clock valuations.

DEFINITION 7 (REGION AUTOMATON) *Given a timed automaton* A*, the corresponding* region automaton *is a finite-state automaton* $R(A) = (S, s_0, R)$ *where* $S = Q \times \Gamma(A)$ *is a finite set of states;* $s_0 = (q_0, \mu_0)$ *is an initial state, where* μ_0 *is the clock region containing the zero clock valuation* v_0*; and* $R \subseteq S \times S$ *is a finite transition relation such that* $((q_1, \mu_1), (q_2, \mu_2))$ *belongs to* R *if and only if either:*

- $q_1 = q_2$*,* μ_2 *is the time successor of* μ_1*, and* μ_2 *satisfies* $I(q_1)$*; or*
- *there exists a discrete transition* (q_1, g, λ, q_2) *such that* μ_1 *satisfies* g*,* $\mu_2 = \mu_1[\lambda = 0]$*, and* μ_2 *satisfies* $I(q_2)$*.*

The *region automaton* captures the behaviors of the original timed automaton exactly, i.e., they satisfy the same sets of properties.

2. Discretization

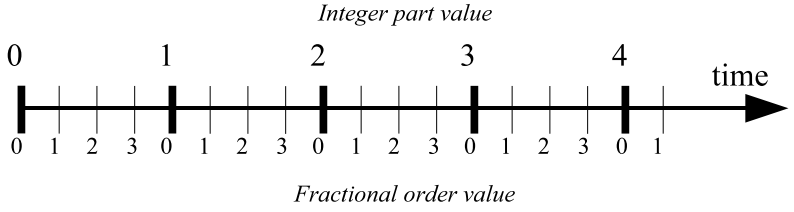
In this section, we give a representation of the region automaton. The representation is implicit as, in the model we construct, time transitions are not enumerated explicitly but are represented by two transitions called the *from-integer* and the *to-integer time transitions*.

Given a timed automaton A, let M_x be the *maximum constant values* in A. For each clock variable $x \in X$, let us introduce two variables: an integer part variable I_x and a fractional order variable F_x.

The *integer part variable* represents the integer part of a clock variable. For a clock variable x, I_x is equal to $\lfloor x \rfloor$ if $x \leq M_x$ and M_x otherwise. Therefore I_x is an integer ranging between 0 and M_x.

For a given clock valuation, order the clock variables that are smaller or equal to the corresponding maximum constant value according to the values of their fractional parts. The *fractional order variable* represents the position of a clock variable in this order. The fractional order variable of a clock variable $x \leq M_x$ is equal to zero if and only if the fractional part of x is equal to zero. For the variables with the smallest, non-zero fractional part (there may be more than one), the corresponding fractional order variable is set to 1. For the variables with the second smallest, non-zero fractional part, the corresponding fractional order variable is set to 2, and so on. If $x > M_x$ then the corresponding fractional order variable F_x is set to 1, as the order between fractional parts is not relevant for clock variables larger than the their maximum constant value. If two clock variables x and y such that $x \leq M_x$ and $y \leq M_y$ have the same fractional part, their fractional order variables are equal. The fractional order variables are integers ranging between 0 and n, where n is the number of clock variables. The order between fractional parts is maintained by the fractional order variables, i.e., given two clock variables x and y such that $x \leq M_x$ and $y \leq M_y$, $F_x \lhd F_y$ if and only if $\langle x \rangle \lhd \langle y \rangle$, for $\lhd \in \{<, =, >\}$. While clock variable $x \in X$ is a real-valued variable, I_x and F_x are discrete (cf. Fig. 2).

Figure 2. The possible values of integer part and fractional order variables. The example shows the case of 3 clock variables with the maximum constant value for the variable shown equal to 4. The fractional order values represent only the relative ordering between fractional parts.



Definition 8 (Discrete Clock Valuations) *Given a set of clock variables X, a* discrete clock valuation *is a function v^d that, for each clock variable $x \in X$, assigns to I_x a value from $\{0, \ldots, M_x\}$ and to F_x a value from $\{0, \ldots, n\}$.*

Let $V^d(X)$ be the set of discrete clock valuations defined for a set of clock variables X. Given a clock valuation $v \in V(X)$, the corresponding *discrete clock valuation* v^d assigns values to each integer part and fractional order variables as described above. Given a clock variable $x \in X$,

we will denote by $v^d(x)$ the pair $(v^d(I_x), v^d(F_x))$, called the discrete value of x.

THEOREM 9 (EQUIVALENCE TO REGIONS) *Each* discrete clock valuation *corresponds to a unique clock region and vice-versa, i.e., given two clock valuation v_1 and v_2, v_1 is equivalent to v_2 (Def. 4) if and only if the corresponding discrete clock valuations v_1^d and v_2^d are equal.*

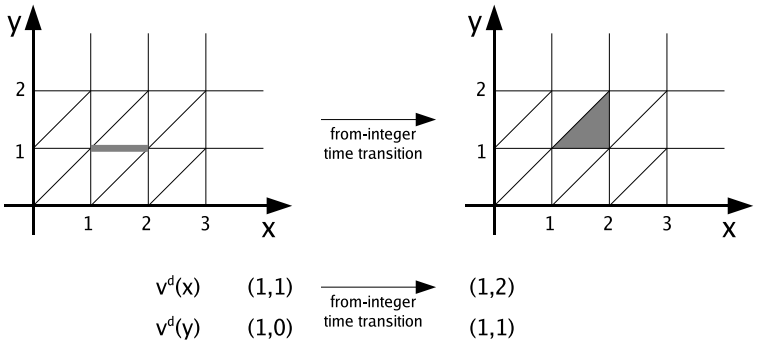
The states of the region automaton can be represented by a pair made of a location and a discrete clock valuation. Now that we have defined discrete clock valuations, a representation for clock regions, and discrete states, we define how transitions between states in the region automaton map to transitions between discrete states in the discrete timed system. The region automaton defines two types of transitions: time transitions and discrete transitions.

Time transitions are represented by two transitions: *(i)* the *from-integer time transition*, which is taken when one of the clock variables has an integer value; and *(ii)* the *to-integer time transition*, which leads to a state where one of the clock variables has an integer value. Each time transition represents a set of actual transitions. We use two types of time transitions to capture two possible scenarios: *(i)* the case where at least one clock variable has an integer value (cf. Figure 3); and *(ii)* the case where none of the clock variables has an integer value (cf. Figure 4). In the figures, the diagrams at the top, represent the clock regions as shaded area as before. The bottom shows a discrete clock valuation by assigning an integer part and a fraction order to each clock variable.

The *from-integer time transition* can be taken only if there exists at least one fractional order variable equal to zero. When this transition is taken, all fractional order variables are incremented by one, while all integer part variables remain unchanged. The example in Fig. 3 contains two clock variables x and y. The maximum constant value of x is 3 and the one of y is 2. A point in one of the diagrams at the top of the figure represents a clock valuation, which assigns the corresponding values to x and y. The thin lines split the clock valuations into regions. A shaded area is used to represent a specific clock region. Initially the discrete value of x is $(1, 1)$ and the one of y is $(1, 0)$. The shaded area in the diagram at the top-left of the figure shows the region corresponding to this discrete state. As time progresses, clock variable y will become greater than 1 before clock variable x reaches 2, and it will have the smallest, non-zero fractional part. Therefore, its discrete value will be $(1, 1)$. At the same time, variable x will still have integer part equal to 1, but its fractional part will become the second smallest one and, therefore,

its discrete value will be $(1, 2)$. The shaded area in the diagram at the top-right of the figure shows the region corresponding to the new state.

Figure 3. Evolution of a region and the corresponding discrete valuation due to the *from-integer time transition*. Each shaded area represents the clock valuations belonging to a region.

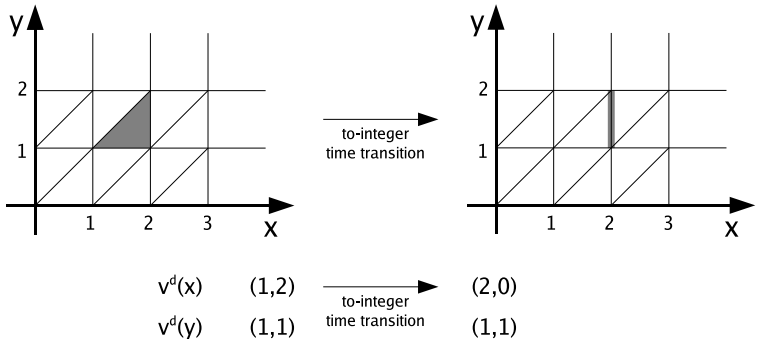


The *to-integer time transition* can be taken only if none of the fractional order variables is equal to zero. The fractional order variables with the largest value (there might be more than one) are set to zero and the corresponding integer part variables are incremented by one. All other integer part and fractional order variables remain unchanged. The example in Fig. 4 contains clock variables x and y as before. Initially the discrete value of x is $(1, 2)$ and the one of y is $(1, 1)$. The shaded area in the diagram at the top-left of the figure shows the region corresponding to this discrete state. As time progresses, variable x will be the first one to reach an integer value, because it has the largest fractional part. Its new value will be $(2, 0)$ and the next value of y will remain $(1, 1)$, since y still has the smallest, non-zero fractional part. The shaded area in the diagram at the top-right of the figure shows the region corresponding to the new state.

The clock variables $x \in X$ such that the integer part variable I_x is equal to M_x and the fractional order variable F_x is greater than zero are treated differently: their integer part and fractional order variables are not updated by the from-integer or the to-integer time transitions. This is because, in the region graph construction, the order between fractional parts is relevant only for those clock variables that are smaller than the corresponding maximum constant value.

Each discrete transition of the region automaton is mapped into a corresponding discrete transition between discrete clock valuations. A discrete clock valuation satisfies a clock constraint g if the corresponding clock region does. Since clock variables are only compared against

Figure 4. Evolution of a region and the corresponding discrete valuation due to the *to-integer time transition*. Each shaded area represents the clock valuations belonging to a region.



integer constants, it is possible to determine if a discrete clock valuation satisfies a clock constraint by looking only at the integer part and fractional order variables. After a transition is taken, the clock variables in the reset set λ must be set to zero. If clock variable x belongs to λ, both the corresponding integer part and fractional order variables are set to zero.

Given a timed automaton A, the result of our discretization is the *discrete timed system* A^d, a system made of two asynchronous processes and containing, for each clock variable x, two discrete variables I_x and F_x. The first process, called the *discrete-transition process* has the same locations and transitions as the original timed automaton, where clock constraints are mapped into expressions over I_x and F_x and reset sets are mapped into resets of these variables, as described above. The second process, called the *time-transition process*, defines the from-integer and to-integer time transitions. The system is modeled using two asynchronous processes: one process defines the time transitions, the other defines the discrete ones. The time transitions can occur at any location of the timed automaton. Having two asynchronous processes allows us to use a smaller representation: time transitions are defined only once but, by virtue of the asynchronous composition, they can be taken at any location of the timed automaton. The idea of separating discrete transitions and time transition into two asynchronous processes has been used by Lamport [11] in his integer discretization based approach for real-time systems. However, as with other integer discretization techniques, this approach handles only integer-valued clock variables and, therefore, does not capture the continuous time semantics of timed automata.

THEOREM 10 (DISCRETE EQUIVALENCE) *Given a timed automaton A, the* discrete timed system *A^d and the region automaton $R(A)$ are equivalent.*

The main advantages of this construction are: *(i)* the construction is implicit, it does not enumerate the clock regions or the time transitions between them; *(ii)* the resulting system can be checked using any of the existing model checking tools and therefore exploit the recent advances in this domain; *(iii)* this approach can easily be extended to the composition of a set of timed automata: since they need to synchronize over the time transitions, we can represent the composition using a *discrete-transition process* for each automaton and a single instance of the *time-transition process.*

3. GOABSTRACTION

The discretization given in the previous section makes it possible to verity properties of timed automata using standard model checking tools. However, in the worst case, the region automaton can be exponential in the number of clock variables and the largest constant. Therefore, even if our construction does not explicitly enumerate the clock regions, model checking might not terminate because of the size of the state space.

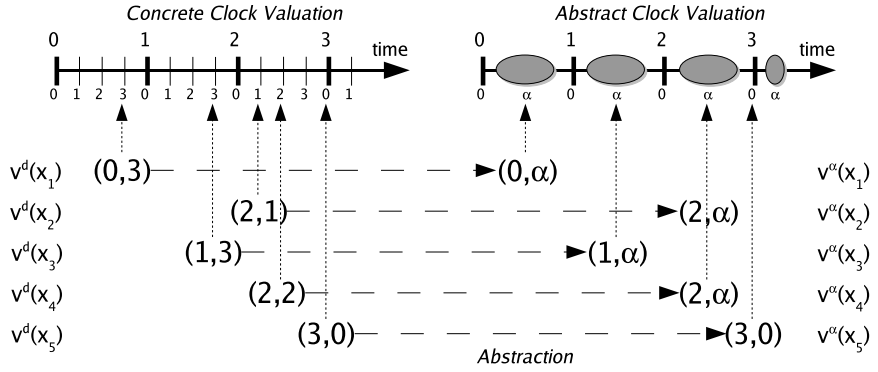
In this section, we introduce a new abstraction technique, called GOABSTRACTION, which aims at reducing the size of the state space. This is a conservative approximation of the behaviors of the system, i.e., each behavior of the original system is maintained in the abstraction, but it may introduce spurious counterexamples.

In the construction given in Section 3, for each clock variable x, the fractional order variable F_x is used to represent the ordering relation between the fractional parts of the different clock variables. Keeping track of this ordering, however, may lead to a number of different permutations that is exponential in number of clock variables. For some applications, this can cause the verification to be intractable.

We propose an abstraction that discards part of the ordering relation between clock variables. In the previous construction, the fractional order variables ranged between 0 and n, where n is the number of clock variables. In the abstraction, we replace the fractional order variables F_x with *abstract fractional order variables* F_x^α. These variables assume values in the abstract domain $F^\alpha = \{0, \alpha\}$, where 0 represents clock variables whose fractional part is equal to zero, and α represents all other possible fractional order values (cf. Fig. 5).

DEFINITION 11 (ABSTRACT CLOCK VALUATIONS) *Given a set of clock variables X, an* abstract clock valuation *is a function v^α that, for each*

Figure 5. The mapping between concrete and abstract clock valuations.



clock variable in $x \in X$, assigns to I_x a value from $\{0, \ldots, M_x\}$ and to F_x^α a value from F^α.

Let $V^\alpha(X)$ be the set of abstract clock valuations defined for a set of clock variables X. Given a clock variable $x \in X$, we will denote by $v^\alpha(x)$ the pair $(v^d(I_x), v^d(F_x^\alpha))$, called the abstract value of x.

Definition 12 (Abstraction Function) *The* abstraction function $h : V^d(X) \to V^\alpha(X)$ *maps discrete clock valuations into abstract clock valuations and is defined as:*

$$h(v^d)(V_x) = \begin{cases} v^d(I_x) & \text{if } V_x = I_x \\ 0 & \text{if } V_x = F_x \text{ and } v^d(F_x) = 0 \\ \alpha & \text{if } V_x = F_x \text{ and } v^d(F_x) \neq 0 \end{cases}$$

Given the abstraction function h, it is possible to construct an abstract timed system A^α using a technique called *existential abstraction* [6]. Existential abstraction produces an over-approximation of the concrete system that is guaranteed to preserve universal CTL (∀CTL) properties. The abstract timed system A^α is analogous to the discrete timed system A^d but uses the abstract fractional order variables instead of the (concrete) fractional order ones. Each transition of A^d is mapped into an abstract transition of A^α as described below.

The *from-integer abstract time transition* can be taken only if there exists at least one abstract fractional order variable equal to 0. When this transition is taken, all fractional order variables equal to 0 are set to α.

The *to-integer abstract time transition* can be taken only if all abstract fractional order variables are equal to α. When this transition is taken,

any non-empty subset of the fractional order variables can be set to 0 and the corresponding integer part variables are incremented by one. Notice that the transitions represented by the *to-integer abstract time transition* can be non-deterministic.

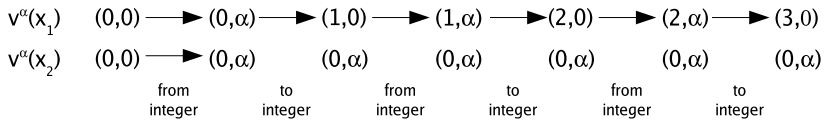
Each discrete transitions is mapped into an abstract discrete transition. The validity of a constraint can be determined by knowing the value of the integer part variables and whether the abstract fractional order variables are equal to zero. The reset of a clock variable can be done by setting both the integer part and the abstract fractional order variables to zero.

Given a timed automaton A, the result of our abstraction is the *abstract timed system* A^α, a system made of two asynchronous processes and containing, for each clock variable x, two discrete variables I_x and F_x^α. The first process, called the *abstract discrete-transition process* has the same locations and transitions as the original timed automaton, where clock constraints are mapped into expressions over I_x and F_x^α and reset sets are mapped into resets of these variables, as described above. The second process, called the *abstract time-transition process*, defines the from-integer and to-integer abstract time transitions.

THEOREM 13 (ABSTRACTION PRESERVATION) *Given a timed automaton A, the* abstract timed system A^α *is an over-approximation of the discrete timed system A^d, i.e., every trace of A^d corresponds to an equivalent trace of A^α.*

The abstraction above, however, is too coarse. Given two clocks that are assigned the same value, it is possible for them to drift apart arbitrarily, i.e., there exists a sequence of abstract time transitions such that the difference between the two clocks grows unboundedly (cf. Fig. 6).

Figure 6. Given two clock variables initially equal, they can drift apart by means of an appropriate sequence of from-integer and to-integer abstract time transitions.



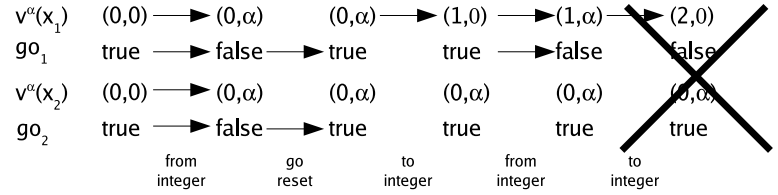
This is because, in the abstraction, we discarded the order between the fractional parts and introduced non-determinism in the to-integer abstract time transition. It is possible to increment one of the clock variables multiple times before incrementing the others.

In order to prevent this and obtain a more precise abstraction, for each of the clock variables $x \in X$, we introduce in our model the Boolean

variable Go_x. The purpose of this variable is to keep track whether a clock variable has been already incremented.

Initially, all Go_x variables are set to *true*. This means that all variables can be incremented. Once a clock variable x has been incremented, the variable Go_x is set to *false*. This prevents the same clock variable from being incremented again. When all Go_x variables are *false*, i.e., every variable has been incremented once, they are set to *true* simultaneously. Fig. 7 illustrates the behavior of the Go_x variables. They guarantee that two clock variables cannot drift apart by more than one time unit, making the abstraction more precise.

Figure 7. The Go_x variables prevent clock variables from drifting apart.



We can now construct the GoAbstraction timed system A^{α}_{Go}, which is obtained by introducing the Go_x variables and updating the abstract transitions.

The *from-integer* GoAbstraction *time transition* is analogous to the from-integer abstract time transition, but it also updates the Go_x variables. If for all clock variables x such that $I_x < M_x$ and $F_x = \alpha$ the corresponding variable Go_x is *false*, then the Go_x variables of all clocks are set to *true*; otherwise the Go_x variables of the clocks whose abstract fractional order variable is equal to 0 are set to *false* and all other Go_x variables are left unchanged. The *to-integer* GoAbstraction *time transition* is analogous to the to-integer abstract time transition, but only clock variables whose Go_x variable is *true* can be updated by this transition.

Given a timed automaton A, the result of GoAbstraction is the GoAbstraction *timed system* A^{α}_{Go}, a system made of two asynchronous processes and containing, for each clock variable x, three discrete variables I_x, F^{α}_x, and Go_x. The first process, called the GoAbstraction *discrete-transition process* has the same locations and transitions as the original timed automaton, where clock constraints are mapped into expressions over I_x and F^{α}_x and reset sets are mapped into setting I_x and F^{α}_x to zero and Go_x to *true*. The second process, called the GoAbstraction *time-transition process*, defines the from-integer and

to-integer GoAbstraction time transitions, and the GoAbstraction reset transition.

Theorem 14 (GoAbstraction Preservation) *Given a timed automaton A, the* GoAbstraction timed system A^{α}_{Go} *is an over-approximation of the discrete timed system A^d, i.e., every trace of A^d corresponds to an equivalent trace of A^{α}_{Go}.*

Moreover, the GoAbstraction timed system is more precise than the abstract time system defined above, that is:

Theorem 15 (Refined Abstraction) *Given a timed automaton A, the* abstract timed system A^{α} *is an over-approximation of the* GoAbstraction timed system A^{α}_{Go}*, i.e., every trace of A^{α}_{Go} corresponds to an equivalent trace of A^{α}.*

4. Experimental Results

In this section, we give some preliminary experimental results that we obtained by applying GoAbstraction to Fischer's mutual exclusion protocol.

This protocol guarantees mutual exclusion by imposing minimum and maximum delays for the execution of some statements. We modeled such delays by means of clock constraints in the timed automaton.

Table 1. Fischer's protocol with symbolic model checking for 4 nodes.

k	*discrete*	*go*
2	28.1s	4.8s
3	82.5s	22.5s
4	175.3s	24.3s
5	355.6s	43.6s
6	728.1s	48.8s

We model checked the protocol using Cadence SMV both as a symbolic model checker and a bounded model checker. The results for symbolic model checking are presented in Table 1. The first column shows the value of the timing parameter k, a parameter of the protocol. The second and third columns report the time required by SMV to perform the verification. The model used for the second column corresponds to the discrete timed system A^d (cf. Section 2) and the one used for the third column corresponds to the GoAbstraction timed system A^{α}_{Go} (cf. Section 3). In both cases, SMV was able to verify mutual exclusion,

which demonstrates that GoAbstraction is precise enough to verify the property. Moreover, by using GoAbstraction, we were able to reduce the running time of the model checker by an order of magnitude.

Table 2. Bounded Model Checking applied to Fischer's protocol with 6 nodes.

k	*discrete*			*go*		
2	100s	l=25	[326s]	19s	l=13	[16s]
3	450s	l=32	[617s]	50s	l=16	[57s]
4	969s	l=38	[2500s]	61s	l=19	[71s]
5	1200s	l=46	[1605s]	137s	l=22	[118s]
6	1800s	l=54	[3115s]	316s	l=28	[347s]

Table 2 shows the results obtained by performing bounded model checking on the same models. Since bounded model checking is mostly aimed at detecting property violations, instead of checking for mutual exclusion, we checked if one of the processes is unable to reach the critical section. Since the protocol is correct, every process is guaranteed to eventually reach the critical section and the model checker reports a counterexample. The first column in the table contains the value of the timing parameter k. The next two columns contain three values: the running time and the depth l at which a valid counterexample was found, and the running time of the verification for depth $l-1$, at which no error can be found. As it can be seen from the results, GoAbstraction has the side effect of reducing the depth at which an error can be detected: this is because all the intermediate steps needed to increment the fractional order variables of the different clocks are removed by the abstraction. Moreover, the running times are reduced again by one order of magnitude.

While these are only preliminary results, they show how, in this case, GoAbstraction is precise enough to prove interesting properties, and it is effective in reducing verification time.

5. Conclusions and Future Work

We described an implicit representation of the region automaton that can be used to perform verification of real-time systems using existing state-of-the-art model checking tools. Since the size of the region automaton can be exponential in the number of clock variables, we introduced GoAbstraction, a new abstraction technique that, by making use of auxiliary variables, is precise enough to preserve interesting prop-

erties of real-time systems. We demonstrated this technique on a typical real-time example.

In our experiments, we manually checked whether a counterexample was spurious. However, this process can be automated, and we would like to do so in future work. While GoAbstraction was sufficient for the example we considered, we would like to develop a counterexample-guided abstraction/refinement framework for timed automata based on GoAbstraction.

Moreover, we would like to develop additional techniques for the verification of real-time systems based on the representation we presented. Specifically, we would like to develop additional abstractions that can be used to address the verification of real-time properties of large systems.

Acknowledgments

This research was sponsored by the National Science Foundation under grant nos. CNS-0411152, CCF-0429120, CCR-0121547, and CCR-0098072, the US Army Research Office under grant no. DAAD19-01-1-0485, the Office of Naval Research under grant no. N00014-01-1-0796, the Defense Advanced Research Projects Agency under subcontract no. SA423679952, the General Motors Corporation, and the Semiconductor Research Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

References

[1] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *Proc. of the 5th Annual IEEE Symposium on Logic in Computer Science*, 1990.

[2] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[3] Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. of the 8th International SPIN Workshop*, 2001.

[4] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In *Proc. of the 15th International Conference on Computer Aided Verification (CAV)*, 2003.

[5] Marius Bozga, Oded Maler, and Stavros Tripakis. Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics. In *Proc. of 10th Conference on Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[6] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[7] David Dill. Timing Assumptions and Verification of Finite-Sate Concurrent Systems. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, 1989.

[8] Aleks Gōllü, Anuj Puri, and Pravin Varaiya. Discretization of Timed Automata. In *Proc. of the 33rd IEEE Conference on Decision and Control*, 1994.

[9] Thomas A. Henzinger and Orna Kupferman. From Quantity to Quality. In *Proc. of International Workshop on Hybrid and Real-Time Systems (HART)*, 1997.

[10] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What Good Are Digital Clocks? In *Proc. of the 19th International Colloquium on Automata, Languages and Programming*, 1992.

[11] Leslie Lamport. Real-Time Model Checking is Really Simple. In *Proc. of 13th Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2005.

[12] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Prof. of the 16th IEEE Real-Time Systems Symposium*, 1995.

[13] Jesper Møller, Henrik Hulgaard, and Henrik Reif Andersen. Symbolic model checking of timed guarded commands using difference decision diagrams. *Journal of Login and Algebraic Programming*, 52-53:52–57, July-August 2002.

[14] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time Systems. In *Proc. of the Workshop on Theory and Practice of Timed Systems*, 2002.

[15] Maria Sorea. *Verification of Real-Time Systems through Lazy Approximations.* PhD thesis, University of Ulm, Germany, 2004.

[16] Stavros Tripakis and Sergio Yovine. Analysis of Timed Systems Using Time-Abstracting Bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001.

[17] Farn Wang. Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems. In *Proc. of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000.

[18] Farn Wang. Region Encoding Diagram for Fully Symbolic Verification of Real-Time Systems. In *Proc. of the 20th Annual International Computer Software and Applications Conference*, 2000.

[19] Farn Wang. RED: Model-Checker for Timed Automata with Clock-Restriction Diagram. In *Proc. of Workshop on Real-Time Tools*, 2001.

[20] Farn Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In *Proc. of the 21st International Conference on Formal Techniques for Networked and Distributed Systems*, 2001.

[21] Sergio Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, December 1997.