Alternating RQ Timed Automata

William K.C. Lam* Robert K. Brayton

Department of EECS, University of California, Berkeley

Abstract. *Two major difficulties in verification with timed automata are state explosion and dependence of complexity on time constants, even with restricted timing constraints. Based on the observation that a vast majority of timed automata have a very regular timing structure, We propose a class of timed automata, alternating RQ timed automata, and prove that they have simple path properties which, even with arbitrary timing constraints, yield efficient verification algorithms. In addition, the complexity of the algorithms is independent of the time constants. Next, we give graphical necessary and sufficient conditions for timed automata to be RQ alternating. Finally, we discuss verification algorithms of alternating RQ L-automata (L-processes).*

1 Introduction

Much research on computer-aided verification has been focused on the sequential aspects of systems with *real-time* abstracted, as evidenced in the verification techniques using CTL's and language containment. Recently, [4] introduced timed automata which augmented the modeling domain of traditional finite state automata to real-time.

A timed automaton is an ω-automaton with timing elements added. To construct a timed automaton from an ω-automaton, we introduce a set of resetable clocks that measure the progress of real-time. A transition of an ω-automaton may have a set of resets of clocks and a set of inequalities (enabling conditions or queries) on the times recorded by the clocks. If a transition has a reset for clock x, then upon the completion of the transition, the value of the clock x becomes zero (reset). If a transition has inequalities involving clock values $x_1, \ldots, x_n$, then the transition is enabled if the inequalities are satisfied by the present values of $x_1, \ldots, x_n$, the present value of x_i being the time elapsed since its last reset of clock i. A formal definition of timed automata can be found in [1].

Example 1. In Fig. 1, the automaton over the alphabet $\{a, b, \$\}$ models a communication receiver using a majority error detection technique. The state S_2 is the accepting state. This timed automaton accepts input sequences that satisfy the following properties: each symbol $\in \{a, b\}$ is repeated three times within 1 unit of time; a message is preceded and ended by a special symbol "$"; the interval between messages is at least 100 units of time. The timed input sequence $\{(\$, 0), (a, 20), (a, 20.3), (a, 20.8), (b, 31), (b, 31.2), (b, 31.4), (\$, 59), (\$, 200), (b, 210.2), (b, 210.5), (b, 211.1), (\$, 232)\}$ is acceptable to the automaton, where the first component is in the alphabet Σ, the second component is the time (real valued) at which the first component occurs. There are three clocks X_a, X_b, X_s which are reset (e.g. $X_a = 0$) or queried (e.g. $X_a < 1$).

* Supported by Fannie and John Hertz Foundation and SRC under contract 92-DC-008.

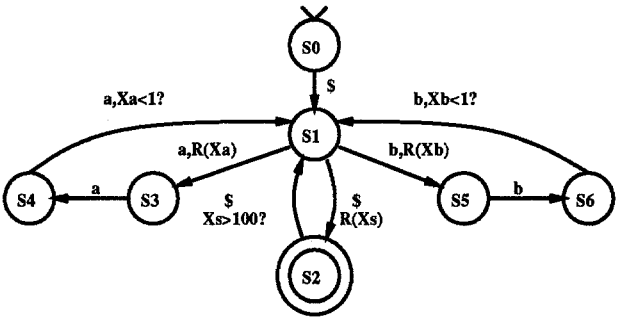

Fig. 1. Real-time Communication Receiver modeled by a Timed Automaton

We denote a query on clock x_i by $Q(x_i)$, a reset on x_i by $R(x_i)$.

2 Traversal in Timed Automata

To verify that a design meets its specifications, the design and the specifications are first expressed with timed automata and then checked whether the language of the automaton of the design is contained in that of the specification automaton. Let D be a design timed automaton, and S, a specification timed automaton. The language of the design automaton is contained in that of the specification automaton, i.e. $L(D) \subseteq L(S)$, if and only if $L(D \otimes S^c) = \phi$, where S^c denotes the complement of S, and $\otimes$ is the production operation. The language of a timed automaton is empty if and only if there is no input string accepted. Thus, verification is intimately related to traversals in timed automata.

Traversal, deciding whether a state is reachable, in a timed automaton consists of two parts. First, decide whether the state is reachable disregarding the timing constraints in the timed automaton. If the state is reachable, then, for each path leading to the state, decide whether the timing constraints along the path are satisfiable. If there is such a satisfiable path, the state is reachable in the timed automaton. It is relatively easy to decide whether a state is reachable disregarding the timing constraints, but it can be very difficult to decide whether there exists a path along which all timing constraints are satisfiable because such a path may not be a simple path (a simple path has no loops). This is illustrated with the following example.

Example 2. In this timed automaton, if K is an integer, then the accepting state S_4 can be reached. And the only way to get from the initial state S_1 to the final state S_4 is to go around the loop K times, i.e. only by traversing a non-simple path. During each visit of the loop, the automaton stays at S_3 for 1 unit of time to get clock X_2 to increment by 1. Thus, to satisfy the timing constraints on the transition between S_2 and S_4, e.g. $X_2 = K$, the loop needs to be traversed K times. If K is not an integer, S_4 is not reachable.

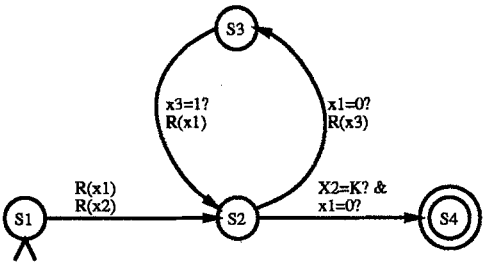

Fig. 2. Time Constant Dependent Traversal

3 Previous Approaches

To tackle traversal in timed automata, [4, 2, 1] introduced the notion of the region graph. A region graph can be regarded as a state graph with each state augmented to include time; that is, a state S now becomes a new state $(S, \Re^n)$, and the timing constraints on transitions serve to partition the new states into equivalent classes of the form $(S, \mathcal{Z})$, where $\mathcal{Z}$ is a zone. [2, 1] give procedures for a coarsest partition compatible with the timing constraints. Once a region graph is constructed from a timed automaton, the effects of the timing constraints are incorporated into the new states; hence, traversal in the timed automaton can be accomplished in its region graph.

The merit of this approach is that it allows arbitrary placement of timing constraints and resets of clocks. The deficiencies are restrictions on timing constraints, e.g. $x < k, x - y < k$, where k is an integer, as well as dependence of the number of augmented states and thus algorithm complexity on the magnitudes of the time constants k.

4 Alternating RQ Timed Automata

In this paper, we observe that many timed automata have a regular pattern of timing constraint placement and propose a class of timed automata that 1) has traversal and verification algorithms independent of time constants and 2) allows arbitrary timing constraints, e.g. $3x - 5y + 4 \leq 0, 4x^3 - 5\log y \leq \sqrt{5y - \frac{1}{z}}$.

4.1 Motivation and Definitions

In placing timing constraints in a automaton, if we want to inquire about a timing status at a certain point on a path via a query, we should have resets for the clocks before the queries on the path. If we want to inquire twice about the same clock, we can use different clocks for each query. This suggests that resets (R's) and queries (Q's) for any clock along any path in a timed automaton should alternate. Although not all timed automata have alternating R's and Q's, timed automata with this property seem intuitively general enough for most applications. For example, the timed automaton in Fig. 1 and all but one timed automata in [4] are alternating RQ timed automata.

Definition 1. 1. The **RQ sequence** of path π, denoted by $\Gamma(\pi)$, is the sequence of resets and queries encountered along π.
2. Given a RQ sequence Γ, the RQ sequence with respect to clock x, denoted by $\Gamma|_x$, is obtained from Γ by deleting all R's and Q's that do not involve x.
3. An RQ sequence Γ is alternating, if, for each clock x, $\Gamma|_x$ has $R(x)$ and $Q(x)$ alternating, and with $R(x)$ initially before $Q(x)$.

Definition 2. An **alternating RQ timed automaton** is a timed automaton[2] with the two following additional properties:

1. For each clock x_i, there is only one pair $R(x_i)$ and $Q(\ldots, x_i, \ldots)$. That is, distinct clocks should be used in measuring different events.
2. For each path π starting from an initial state, $\Gamma(\pi)$ is alternating, i.e. $\Gamma(\pi)|_{x_i}$, is alternating for each x_i.

4.2 Scope of Alternating RQ Timed Automata

We examine how restricted is this class of timed automata. In general, a timed automaton can have several resets and queries for a clock; thus, it is interesting to determine the class of timed automata that can be transformed into alternating RQ timed automata. Unfortunately, not all timed automata with multiple resets for each clock can be transformed to satisfy condition 1 of alternating RQ timed automata. However, all timed automata with a single reset and possibly multiple queries for each clock can be transformed to satisfy condition 1 as stated in the following theorem.

Theorem 3. *Every timed automaton with a single reset for each clock can be transformed to satisfy the alternating RQ condition 1 while preserving its language.*

Proof. Let x be a clock in a timed automaton M such that there is only one reset for x, $R(x)$ and there are m $Q_i(x)$'s on edges $q_1, ..., q_m$. The transforming procedure is as follows. Replace $R(x)$ by $\{R(x_1), ..., R(x_m)\}$, and $Q_i(x)$ on q_i, by $Q_i(x_i)$. Now there is only one pair of R(x_i) and $Q_i(x_i)$ for each i. Repeat the above transformation for all clocks with multiple Q's.

Now it remains to show that the transformed automaton accepts the same language as the original one. Denote the transformed automaton by M'. Since the transformation changes only the resets and queries, we need only show that the set of inequalities induced by any input timed sequence α on M' is satisfiable if and only if the corresponding set on M is satisfiable. We show, however, the set of inequalities induced by $Q_i(x_i)$ on edge q_i of M' is the same as the set induced by Q_i(x) on q_i of M, because the only difference between $Q_i(x_i)$ in M' and $Q_i(x)$ in M is the renaming of variables. ■

Transformation of timed automata with multiple resets of the same clock to satisfy the alternating RQ condition 1 is much more complicated. Whether it can be done may depend, for example, on whether removal or addition of certain timing constraints affects

[2] Here, arbitrary timing constraints are allowed

the original specification the designer has in mind. Techniques to achieve it may involve introduction of new states and relocation of timing constraints. A general technique is not known at this time.

Now we try to transform a timed automaton satisfying condition 1 to also satisfy condition 2. This is not always possible. The reason that an automaton may satisfy condition 1 but not 2 is that there is a path π and a clock x such that $\Gamma(\pi)|_x$ is not alternating. Specifically, there is a loop such that only $R(x)$ or $Q(x)$ is in the loop.

In the case where only $Q(x)$ is in the loop and $Q(x)$ involves comparisons with time constants only, i.e. $x \diamond k$ where k is a constant and $\diamond \in \{\leq, <, \geq, >, =\}$, we can eliminate the loop and convert the automaton to satisfy both conditions. This is because, in a real system, a transition takes a finite amount of time to complete. Thus, if this finite transitional time is also modeled, then each time the loop is traversed, the value of the clock x is increased by a finite amount. So, after a finite number of transitions, x is larger than the maximum time constant in $Q(x)$, making $Q(x)$ settle to a constant value, 1 or 0; that is, $x \diamond k$ becomes true or false. Thus, we can expand the loop with $Q(x)$ in the timing specification with a finite number of new states and clocks, and remove $Q(x)$ from loops. Then, we can apply the transformation in Theorem 3 to convert it to satisfy condition 1. Hence, with proper modeling, an automaton satisfying condition 1 and having cycles with only $Q(x)$'s of comparison type $x \diamond k$ can be made to meet condition 2[3].

In the case of only $R(x)$'s in a loop or multiple $R(x)$'s, it is not always possible to transform the automaton to satisfy condition 2 without losing some timing specifications. In this case, the designer may need to rearrange the R's and Q's while retaining his intended specifications. The example below illustrates one case where being not RQ alternating indicates an incomplete specification, while a complete and correct specification turns out to be RQ alternating.

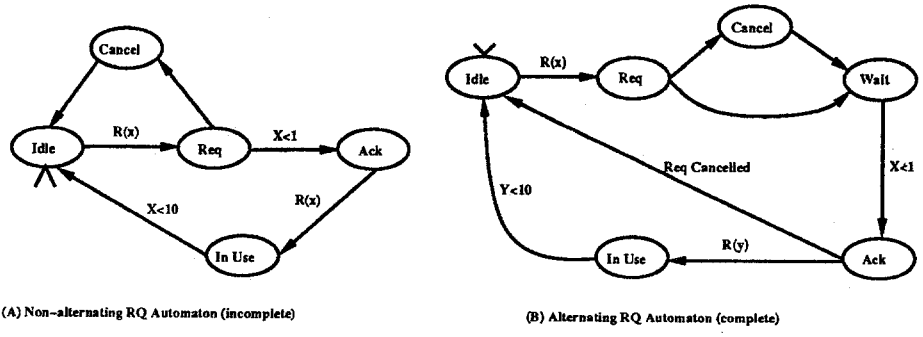

Fig. 3. Specifying a Bus Controller

[3] Of course, in doing this transformation, we introduce new states; the number of which depends on the time constants; hence our claim about complexity independent of time constants disappears.

Example 3. Fig. 3(a) is an automaton modeling a bus-requesting protocol. When a system is not requesting the use of a bus, the automaton is in state "idle". When the system requests the use of a bus, it enters the state "req" and requires the arbiter to grant a bus within 1 unit of time, as specified by the timing constraint "$x < 1$". If a bus is granted, the system will have at most 10 units of time over the control of the bus, as specified by the timing constraint "$x < 10$". In addition, the system can cancel a request any time before a bus is granted, thus entering the state "cancel". This automaton is not RQ alternating because the loop "$idle \rightarrow req \rightarrow cancel \rightarrow idle$" contains only a reset without a query. Now, let us see how to modify this to be RQ alternating while retaining the essentials of the specifications.

A close examination of the specification by the automaton in Fig. 3(a) reveals a flaw; the automaton does not model the behavior of the arbiter in case the system cancels a request, i.e. the arbiter may try to allocate a bus even when it is canceled. So a more complete specification should include this case. And we hope that this inclusion will produce a RQ alternating timed automaton. It may be reasonable to assume that the arbiter will respond faster when the system cancels its request, because the arbiter does not have to wait for available buses to respond; hence, it is not too restricting to require the arbiter to respond within 1 unit of time in the case of cancellation. By adding this reasonable constraint, we obtain the automaton shown in Fig. 3(b) which is an alternating RQ timed automaton.

4.3 Graphical Necessary and Sufficient Conditions

Here we examine the placement characteristics of timing constraints in alternating RQ timed automata. Knowing these characteristics facilitates modeling systems with alternating RQ timed automata, deciding whether a timed automaton is RQ alternating, and studying effects of arbitrary placements of timing constraints.

Definition 4. In a graph, an edge e is a **cut** for an ordered vertex pair (v_1, v_2) if either v_2 is not reachable from v_1 or the removal of e from the graph makes v_2 unreachable from v_1. Denote the cut by $e|(v_1, v_2)$.

Thus, if $e|(v_1, v_2)$, then all paths from v_1 to v_2 must pass through e.

Since condition 1 is easy to check, the main concern is on the placement of timing constraints to meet condition 2. Suppose we want to place a $R(x)$ on edge e_r and a $Q(x)$ on e_q. What is the graphical relationship between e_r and e_q such that all paths from the initial states have $R(x)$ and $Q(x)$ alternating? Condition 2 requires that $R(x)$ be encountered before $Q(x)$; this translates into the rule that all paths from the initial states to e_q must pass e_r. Condition 2 further requires that $R(x)$ and $Q(x)$ alternate thereafter; this means all loops from e_r to itself must also pass through e_q and similarly for e_q. This intuition is formally stated in the following theorem.

Theorem 5. *Assuming a given timed automaton satisfies the alternating RQ condition 1 (which can be easily checked), it is an alternating RQ timed automaton, i.e. it satisfies condition 2, if and only if for each clock x,*

$$e_r|(S_0, v_q),\ e_r|(u_q, v_q),\ e_q|(u_r, v_r).$$

where S_0 is any initial state, $e_r = (v_r, u_r)$ is the edge where $R(x)$ resides, and $e_q = (v_q, u_q)$, where $Q(x)$ resides.

Proof. Necessity. First, if e_r is not a cut for (S_0, v_q), then there is a path from S_0 to v_q without passing e_r. This path encounters $Q(x)$ before $R(x)$, violating condition 2. Second, if e_r is not a cut for (u_q, v_q), then there is a path from u_q to v_q without passing e_r. Then, the loop consisting of the path and $e_q = (v_q, u_q)$ contains $Q(x)$ but not $R(x)$; thus, traversing the loop several times produces several consecutive $Q(x)$, violating the RQ alternating graph. Similarly, if e_q is not a cut for (u_r, v_r), then there is a loop containing only $R(x)$ which can be traversed consecutively to violate condition 2.

Sufficiency. First, $e_r|(S_0, v_q)$ guarantees that $R(x)$ be encountered before $Q(x)$ if $Q(x)$ is ever encountered along any path from an initial state S_0. Second, $e_r|(u_q, v_q)$ guarantees that if there is a loop containing $Q(x)$ then the loop also contains $R(x)$; thus, between any two $Q(x)$'s along any path there is at least one $R(x)$'s. Similarly, $e_q|(u_r, v_r)$ guarantees at least one $Q(x)$'s between any two $R(x)$'s along any path. Together, $e_r|(u_q, v_q)$ and $e_q|(u_r, v_r)$ assure that $R(x)$'s and $Q(x)$'s alternate along any path. Combining with $e_r|(S_0, v_q)$, alternating RQ condition 2 is satisfied. ■

The following theorem provides a sufficient condition that can be checked by inspection.

Theorem 6. *A timed automaton, represented by a graph G, is RQ alternating if, for each clock x, the removal of the edges of $R(x)$ and $Q(x)$ ($e_r = (v_r, u_r)$ and $e_q = (v_q, u_q)$, respectively) partitions G into two disjoint subgraphs G_1 and G_2 such that v_r, u_q, and all initial states are in G_1, while u_r, and v_q are in G_2. See Fig. 4.*

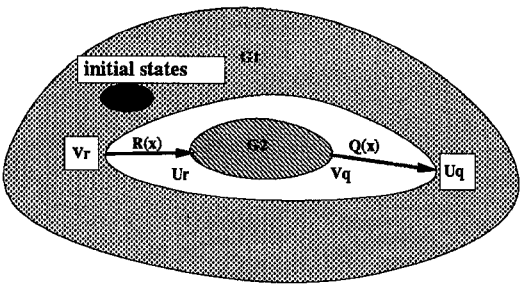

Fig. 4. Illustration for Theorem

Proof. All paths (if any) from an initial state to e_q must pass e_r; thus, $e_r|(S_0, v_q)$. Then, all paths (if any) from u_q to v_q must pass e_r; thus, $e_r|(u_q, v_q)$. Finally, all paths (if any) from u_r to v_r must pass e_q; thus, $e_q|(u_r, v_r)$. By Theorem 5, the timed automaton is RQ alternating. ■

4.4 Modeling Capacity

Roughly speaking, in view of Theorem 6, an RQ pair in an alternating RQ timed automaton constrains a set of partial paths having common head edges and common tail edges, for example, the partial paths in the subgraph G_2 in Fig. 4. Thus, to impose a timing constraints on a set of partial paths, auxiliary states may be introduced to make the partial paths share common head edges and common tail edges.

Alternating RQ timed automata can not model all timing behaviors; and a characterization of the behaviors modelable by a general, but not an alternating RQ, timed automaton is beyond the scope of this paper. An advantage of alternating RQ timed automata is that the placement of the timing constraints reveals easily the paths being constrained. It is often more difficult to see how to do this for non-alternating RQ timed automata — thus, non-alternating RQ timed automata are often difficult to place timing constraints to meet the exact specifications without constraining other paths needlessly. Further, most timed automata encountered so far are either RQ alternating or can be converted, for example, all those in [5, 3] and all but one in [4].

4.5 Composition of Alternating RQ Timed Automata

It is very common that large systems are built from a collection of smaller systems. We show that composing alternating RQ timed automata preserves the RQ alternating property, under a very general definition of composition.

Definition 7. Given a collection of alternating RQ timed automata, which may have outputs, we define an **I/O composition** of the collection to be the timed automaton formed by interconnecting the inputs and outputs of the automata in the collection, e.g. inputs of an automaton may be connected to the outputs of another. The set of initial states of the composition automaton is the Cartesian product of the initial states of the sub-automata in the collection. The sets of clocks of any two sub-automata are considered distinct.

The I/O composition includes the usual notion of product of automata, because a product automaton can be formed by simply connecting together the inputs of the sub-automata.

Theorem 8. *The timed automaton derived from an I/O composition of a collection of alternating RQ timed automata is RQ alternating.*

Proof. Any path in the composition automaton consists of paths in some of the sub-automata in the collection. Because all the sub-automata are RQ alternating, the RQ sequences of all the paths starting from initial states in the sub-automata are alternating; therefore, the RQ sequence of any path starting from an initial state in the composition automaton is also alternating. Further, because the sets of clocks of any two sub-automata are considered distinct, there is only one pair $R(x)$ and $Q(\ldots, x, \ldots)$ for each clock x in the composition automaton. Therefore, the composition automaton is RQ alternating. ■

There are situations where only component automata of an automaton are of concern. Then, will an alternating RQ timed automaton imply its sub-automata RQ alternating? In general, this is not true, because an automaton may contain a sub-automaton with unreachable paths whose RQ sequences are not alternating. Example 4 shows an alternating RQ automaton with a non-RQ alternating sub-automaton.

Example 4. In Fig. 5, M_3 is an I/O composition of M_1 and M_2, in which the outputs of M_1 are fed to the inputs of M_2. M_3 is RQ alternating, but M_2 is not. This is because the paths in M_2, e.g. $r_1, r_2, r_3, r_2, r_3, \ldots$, that have non-alternating RQ sequences are not reachable in the composition automaton M_3.

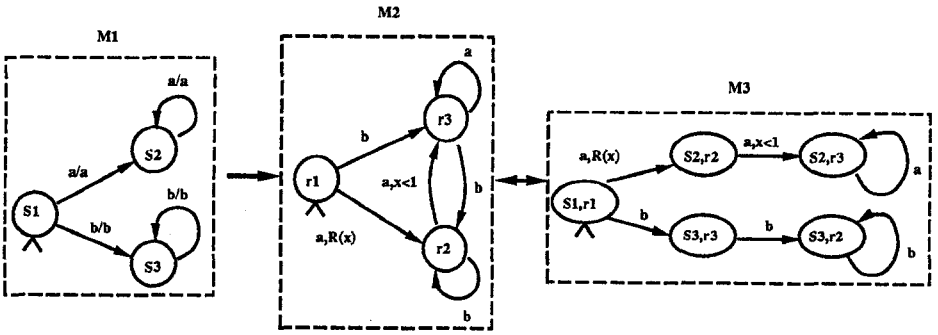

Fig. 5. Composition of RQ Automata

Thus, if every path in a sub-automaton is reachable in the composition automaton, then the RQ alternating property of the composition automaton will be inherited to the sub-automaton. This is because if the sub-automaton is not RQ alternating, then the input sequence to the composition automaton, which activates the non-alternating path in the sub-automaton, will cause a non-alternating RQ sequence in the composition automaton. A useful special case is when the I/O composition is a production in the the usual convention.

Theorem 9. *If a product automaton is RQ alternating, then all of its component automata are also RQ alternating.*

Proof. A product automaton can be formed by connecting the inputs of all its component automata together. Thus, every reachable path in a component automaton is also reachable in the product automaton. If a component automaton is not RQ alternating, then the paths with non-alternating RQ sequences can be made reachable from the product automaton, causing the product automaton to be non-RQ alternating. ■

4.6 Simple Path Properties

Here we prove some properties of alternating RQ timed automata that make reachability analysis independent of time constants and also simplify verification algorithms. As seen

in example 2, reachability analysis can be quite complicated in general timed automata — some states can only be reached through non-simple (with loops) paths whose lengths depend on the timing constraints. Further, there exist examples in which some loops can only be traversed a finite number of times, thus, making language acceptance testing difficult for acceptance conditions which involve infinite looping, e.g. Buchi, Muller, and L-automata. In contrast, alternating RQ timed automata avoid these complications — a state is reachable from another if and only if it is reachable through a simple path, and a loop can be traversed infinitely often if and only if it can be traversed once. These properties are proved in the following theorems.

Definition 10. 1. Assume automata take no time to complete transitions; thus, the time spent along a path is the sum of times spent at the states on the path. Let μ_i, **interarrival variable**, be the time spent on state i.
2. Given an RQ sequence Γ, each query induces a set of inequalities on interarrival variables, μ_i's. Denote by $\Theta(\Gamma)$ the set of inequalities induced by this RQ sequence.
3. In timed automata, path π is **traversable** if $\Theta(\Gamma(\pi))$ is satisfiable. State v is **reachable** from state u if there is traversable path from u to v.

Theorem 11. *In alternating RQ timed automata, a state is reachable from another state if and only if it is reachable through a simple path.*

Proof. Assume traversable path π from v to u has a loop, i.e. $\pi = v, ..., v_l^1, ..., v_l^2, v_k, ..., u$, where $v_l^1, ..., v_l^2$ is the loop and v_l^1 denotes the state v_l being visited at the beginning of the loop while v_l^2, at the end of the loop. v_l^1 and v_l^2 are the same state, the superscripts denote the order of visit. Let $\pi' = v, ..., v_l^1, v_k, ..., u$ be derived from π by deleting the loop. Consider the sets of inequalities induced by π and π'. The set of inequalities induced by π consists of three subsets of inequalities, Θ_1, Θ_2, and Θ_{loop}. Θ_1 is induced by Q's on the path $v, ...v_l^1$; Θ_{loop}, on the path $v_l^1, ..., v_l^2$; Θ_2, on the path $v_l^2, ..., u$. Similarly, the set of inequalities induced by π' consists of two subsets of inequalities, Θ_1' and Θ_2'. Θ_1' is induced by Q's on the path $v, ..., v_l^1$; Θ_2', on the path $v_l^1, ..., u$. Note that Θ_1' is the same as Θ_1 except for renaming of interarrival variables. Let λ be the sum of the interarrival variables of the states from v_l^1 to v_k on π, i.e. the time spent in the loop, λ', from v_l^1 to v_k on π'. Now consider the $Q(x)$'s that induce inequalities in Θ_2. The simplest case is when the $Q(x)$'s $R(x)$ occurs after the loop, then the inequalities induced by this $Q(x)$ on both Θ_2 and Θ_2' are the same except for the renaming of variables. Consider now the case where the $Q(x)$'s $R(x)$ occurs before or in the loop. Because of the alternating RQ condition, the $Q(x)$'s $R(x)$ cannot occur in the loop, but before the loop; thus, if some inequalities in Θ_2 contain some interarrival variables of states in the loop, then these inequalities contain the interarrival variables of all the states in the loop; that is, the inequalities in Θ_2 contain only the interarrival variables of states before and after the loop plus λ, not interarrival variables of a subset of states in the loop. Similarly, the variables of Θ_2' consist of the interarrival variables between v and v_l^1, between v_k and u, and λ'. Again, Θ_2' is the same as Θ_2 except for renaming of interarrival variables and identifying of λ with λ'. Therefore, the inequalities $\{\Theta_1', \Theta_2'\}$ are a subset of $\{\Theta_1, \Theta_2, \Theta_{loop}\}$. Because path π is traversable, $\{\Theta_1, \Theta_2, \Theta_{loop}\}$ is satisfiable; thus, $\{\Theta_1', \Theta_2'\}$, a subset, is also satisfiable; Therefore, the simple path π' is also traversable. ■

Theorem 12. *In alternating RQ timed automata, a loop is traversable infinitely often if it is traversable once.*

Proof. If a $Q(x)$ is in the loop, then the corresponding $R(x)$ must appear somewhere before $Q(x)$ but also in the loop. Because, otherwise, a path going through the loop twice will generate two consecutive $Q(x)$'s, violating the alternating RQ condition. So the set of inequalities induced by going through the loop twice consists of two identical subsets of inequalities, the subset being those induced by going through the loop once. If the subset is satisfiable, the set made of the subset is also satisfiable. Therefore, once-traversability implies infinite-traversability. ■

Since in an alternating RQ timed automaton a state is reachable from another state if and only if it is reachable through a simple path, reachability can be checked by examining all simple paths between the two states. For each such simple path, the inequalities resulted from the timing constraints on the path can be checked for feasibility by using, for example, linear programming for linear constraints. Therefore, reachability analysis is independent of time constants, and arbitrary timing constraints are allowed (nonlinear constraints can be checked using a general nonlinear solver.) Examining all simple paths is not practical, the following section proposes an algorithm that uses these simple path properties in verification without checking explicitly all simple paths.

5 Verification with Alternating RQ Timed Automata

In this section we consider L-automata and L-processes that have timing constraints on some transitions and are RQ alternating. We propose algorithms to check for language containment.

5.1 L-automata and L-processes

Here we briefly review L-automata and L-processes. For a more detailed discussion, the reader is referred to [8].

An L-automaton is a 4-tuple

$$\Gamma = (M_\Gamma, I(\Gamma), R(\Gamma), Z(\Gamma))$$

where M_Γ is the transition matrix, $I(\Gamma)$ is the set of initial states where $\phi \neq I(\Gamma) \subset V(\Gamma)$, $V(\Gamma)$ is the set of vertices of Γ, $R(\Gamma) \subseteq V(\Gamma) \times V(\Gamma)$ is the set of "recur" edges of Γ, $Z(\Gamma) \subseteq 2^{V(\Gamma)}$ is the set of "cycle" sets of Γ. A sequence of states $v = (\nu_0, \nu_1, \ldots)$ is accepted if either for some integer N and some $C \in Z(\Gamma)$ $\nu_i \in C$ $\forall i > N$ or the set $\{i : (\nu_i, \nu_{i+1}) \in R(\Gamma)\}$ is infinite. In words, a sequence of states is accepted is either the sequence visits eventually only the states in some cycle set C or it visits some recur edges infinitely often. An L-automaton is deterministic if M_Γ produces a unique next state for a given input alphabet at any present state. An L-process is similar to an L-automaton except it has an output function and the acceptance conditions are interpreted complementarily. An L-process is a 5-tuple

$$\Gamma = (M_\Gamma, O(\Gamma), I(\Gamma), R(\Gamma), Z(\Gamma))$$

where $O(\Gamma)$ is the output function with the state space as its domain. A sequence of states $(\nu_0, \nu_1, \ldots)$ is rejected, rather than accepted as in L-automata, if either for some integer N and some $C \in Z(\Gamma)$ $\nu_i \in C$ $\forall i > N$ or the set $\{i : (\nu_i, \nu_{i+1}) \in R(\Gamma)\}$ is infinite.

To verify that a system accomplishes a task, the system is represented by an L-process while the task is represented by a deterministic L-automaton. If the language of the L-process, the set of all accepting input sequences, is a subset of that of the L-automaton, the system is said to accomplish the task. To determine language containment, one searches for an input sequence that is accepted by the L-process but not by the L-automaton. If the search is successful, the language containment fails. To begin, after determining all reachable states, all recur edges in the L-process and the L-automaton are removed so that the sequences rejected by the L-process or accepted by the L-automaton by recurring on their respective recur edges are eliminated; these sequences do not contribute to the determination of the language containment. Then, a product machine of the L-process with the L-automaton is formed and the strongly connected components of the graph of the product machine are identified. If there is a strongly connected component (a subset of vertices) that is not a subset of some cycle set of either the L-process or the L-automaton, the containment fails, because this strongly connected component contains a cycle that is accepted by the L-process (by not being in some cycle set of the L-process) and is rejected by the L-automaton (by not being in some cycle set of the L-automaton). Although some verifiers use implicit state enumeration techniques instead of this explicit graphical method [11, 9], the essence is similar.

5.2 Verification with Alternating RQ L-automata

Here we consider language containment of L-automata (L-processes) whose resets and queries alternate. Because the effects of timing constraints in timed automata are to prevent some paths from being traversable, we construct a constraining automaton to characterize all the untraversable paths; thus, the behavior of the original timed automaton is completely captured by the original timed automaton ignoring the timing constraints and the constraining automaton. If this constraining automaton is constructed, a generic verifier can be used to verify properties specified by the original timed automaton. In general timed automata, these constraining automata can be too complicated to be practical due to traversal complications in timed automata. However, in alternating RQ timed automata, the traversabilities of paths are better structured, e.g. the simple path properties, thus, facilitating construction of constraining automata.

Definition 13. 1. An ω **path** is an infinite path from some initial state. An ω **simple path** is a path consisting of a simple path from an initial state to a simple cycle.

2. For a given path π, the **simplified path(s)** of π, denoted by $S(\pi)$, are the set of simple path(s) derived from π by deleting intermediate loops. For example, if $\pi = (v_1, v_2, v_3, v_4, v_2, v_4, v_3)$, then $S(\pi) = \{(v_1, v_2, v_4, v_3), (v_1, v_2, v_3)\}$, which is a set of simple paths.
3. A minimal RQ sequence is an unsatisfiable RQ sequence such that removal of any R's and Q's from the sequence makes it satisfiable.

4. Since, in alternating RQ timed automata, R's and Q's in any RQ sequence appear pairwise, any RQ sequence can be represented by the Q's alone. For the set consisting of all minimal RQ sequences of all ω simple paths, call the Q's in the set the **minimal blocking Q set**, denoted by X_Q.

Theorem 14. *Let X_π be the set of all untraversable ω simple paths in alternating RQ L-automaton L, L^u, the automaton of L with timing constraints ignored. Then, there is a traversable ω path in L if and only if there is a ω simple path in $L^u - X_\pi$.*

Proof. If direction. A ω simple path in $L^u - X_\pi$ is traversable in L.

Only if direction. If there is a traversable path in L, a simplified path of the traversable path is traversable by the simple path properties of alternating RQ timed automata. This simplified path is therefore not in X_π, thus in $L^u - X_\pi$. ■

This theorem says that if we can construct an automaton to represent X_π, then timing verification on L can be done using the ordinary automaton L^u and the automaton representing X_π.

5.3 Construction of Constraining Automata

X_π can be represented by representing the edges where the Q's of X_π reside, because it is the Q's that determine X_π. And the number of Q's in X_π can be reduced by using the minimal block Q set, X_Q. So, the problem of representing X_π is reduced to that of representing sequences of edges.

Let $\{e_i = (v_i, u_i), 1 \leq i \leq n\}$ be a sequence of edges on which Q's of a minimal RQ sequence reside. For a ω simple path $(v_1, \ldots, v_r, \ldots, v_r)$, the simple cycle $v_r, \ldots, v_r$ is "cut" so that the ω simple path is regarded as a plain simple path $(v_1, \ldots, v_r, \ldots, v_r)$ (In fact, the alternating RQ requirement implies that a ω simple path can be split into two subpaths $(v_1, \ldots, v_r)$ and $(v_r, \ldots, v_r)$ which can be considered separately). The constraining automaton has an initial state v_0 which is identified with any initial state in the original timed automaton and an exclusion state X. Start with e_1. Create states v_1 and u_1 and a transition from v_0 to v_1 with the input alphabets on the transition being the union of strings along all simple paths from v_0 to v_1 without passing any e_i. One way to obtain this union is to remove all edges e_i $i \geq 2$ and find the regular language with v_1 being the only accepting state and delete all Kleen closures from the language. Then, create another transition from v_1 to u_1 with the same transition alphabet as that in the original automaton. For e_i, create states v_i and u_i, add transition from v_i to u_i with transition alphabet being the union of strings along all simple paths from u_{i-1} to v_i without passing any e_i. Then add a transition from v_i to u_i with the same alphabet as the original automaton. Identify u_n with the exclusion state X. Finally, a self-loop at X is added. A constraining automaton for several sequences of edges is just the product of the constraining automaton for each sequence. The product automaton can be simplified by any state reduction technique, e.g. the state minimization algorithm in [10].

When the timed automata are nondeterministic, there may be more than one distinct paths having the same sequence of input alphabets and different timing constraints. Then an extra step needs to be included in the above algorithm for constructing constraining

automata. By including the input strings in the constraining automata, all paths with the input strings will be eliminated; therefore, if, among the paths having the same input strings but different timing constraints, there are some paths that are traversable and some untraversable, then all these paths should not be included in the constraining automata. Of course, if all these paths are untraversable, then all these paths should be included in the constraining automata as mentioned above. In summary, the extra step is as follows. For a set of input strings leading to a untraversable path, check all the paths with the same set of input strings for timing constraints. If all these paths are untraversable, include all these paths in the constraining automaton as before. If some of these paths are traversable, do not include any of these paths in the constraining automaton.

Once a constraining automaton L_x for X_π is constructed, $L^u - X_\pi$ is just the product machine of L^u and L_x with the modified cycle sets Z' as: $Z' = \{C_i'\}, C_i' = \{(c, y) : c \in C_i, y \notin X\}$. Therefore, an accepting ω path will never enter the exclusion state X, avoiding all the untraversable simple paths. Hence, all accepting ω simple paths in $L^u \times L_x$ are $L^u - X_\pi$.

Example 5. Fig. 6 is an example of how to construct a constraining automaton to represent timing behaviors. The partial path "$S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5$" is the only untraversable partial path; and the minimal blocking Q set is $\{x > 2, y < 1\}$ with the Q-edges being $\{(S_3, S_4), (S_4, S_5)\}$. The constraining automaton enters into the exclusion state S_5' when $(S_3, S_4), (S_4, S_5)\}$ are traversed. The strings (of simple paths) from the initial state S_0 to S_3 are $\{acb, bacb\}$, which are the transition strings for the transition from S_0' to S_3' in the constraining automaton. The rest of the construction is straightforward. The constraining automaton is shown in Fig. 6(b). In fact, only (S_4, S_5) needs to be represented because (S_3, S_4) is traversed whenever (S_4, S_5) is. The simplified automaton is shown in Fig. 6(c).

By representing the Q-edges, a **subset of paths** are represented, not just one. When an unsatisfiable **partial path** is represented by its Q-edges, all paths through the partial path are represented.

Note that finding X_π involves linear programming; thus, arbitrary timing constraints are allowed and the verification algorithm is independent of timing constraints.

Besides the method of constructing an ordinary automaton from the original timed automaton and then proceeding to verify on this untimed automaton, another method is to start verification with timing constraints ignored. If and when it fails, timing constraints are added to eliminate "bad" paths. This is repeated until either all bad paths are eventually eliminated by the timing constraints, whence the containment check succeeds, or an example of bad behavior meeting all timing constraints is found, e.g. [5, 3].

The scheme is as follows. We have an ordinary containment checker which ignores all timing constraints, a linear program to check for satisfiability of linear timing constraints, and a routine to construct an automaton which excludes the bad paths not traversable due to the timing constraints. We start with the automaton with timing constraints ignored. If the containment check fails, the checker will produce a simple bad path leading to a bad cycle, which will be checked for traversability with the timing

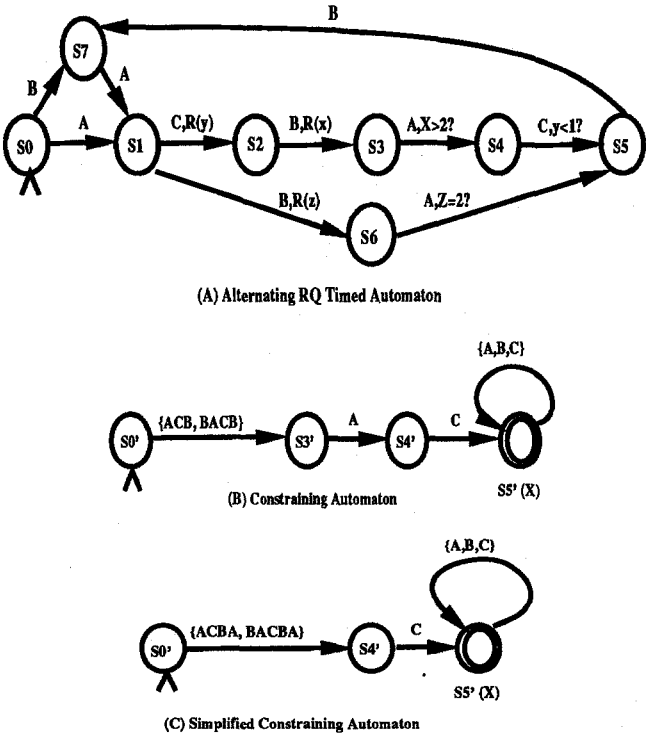

Fig. 6. Construction of a Constraining Automaton

constraints. If either all bad paths or the bad cycle are not traversable due to the unsatisfiability of the timing constraints along the paths, these bad paths and cycle do not really exist in the timed automaton. Thus, a constraining automaton is created to represent the Q's of the bad path. Note that this constraining automaton will eliminate *all* paths going through the unsatisfiable Q's, not just the bad path. This new automaton is then combined with the original automaton so that the resulting product automaton will not have these bad paths and bad cycle. The above procedure is repeated on this new product automaton.

If the above procedure is performed on a general timed automaton, all paths, not only the simple paths, need to be examined; there are infinitely many non-simple paths. Further, it is not obvious how to check infinite traversability of loops; loops that can only be traversed a finite number of times can be constructed. With alternating RQ timed automata, Theorem 11 guarantees that only simple paths need to be checked. Theorem 12 guarantees that loops need only be checked for traversability around the loop once in order to check for infinite traversability. Therefore, the ordinary containment checker needs to provide only the simple bad paths and simple bad cycles to the linear program.

6 Conclusion

In this paper, we proposed alternating RQ timed automata, whose resets and queries alternate along any path from an initial state, and showed that this class of timed automata have the simple path properties: a state is reachable if and only if it can be reached via a simple path and a loop can be traversed infinitely often if it can be traversed once. With these properties, timing verification algorithms have complexities independent of timing constraints and allow arbitrary timing constraints. Next, we observed that this class of timed automata contained most examples of timed automata to date. Then, we gave graphical necessary and sufficient conditions for timed automata to be RQ alternating. Finally, we discussed a verification strategy using alternating RQ L-automata.

References

1. R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification base on automata emptiness. *IEEE Real-Time Systems Symposium*, 1992.
2. R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. *International Conference on Computer-Aided Verification*, 1992.
3. R. Alur, A. Itai, R. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *International Conference on Computer-Aided Verification*, 1992.
4. Rajeev Alur and David Dill. Automata for modeling real-time systems. *1990 ACM International Workshop on Timing Issues In the Specification and Synthesis of Digital Systems*, 1990.
5. Felice Balarin and Alberto Sangiovanni-Vincentelli. A verification strategy for timing constrainted systems. *International Conference on Computer-Aided Verification*, 1992.
6. E. Clarke, O. Grumberg, and R. Kurshan. A synthesis of two approaches for verifying finite state concurrent systems. *Workshop on Automatic Verification Methods for Finite State Systems*, 1989.
7. R.Kurshan E.M.Clarke, I.A.Draghicescu. A unified approach for showing language containment and equivalence between various types of ω-automata. *Tech. report, CMU,*, 1989.
8. Z. Har'El and R. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, Jan. 1990.
9. R. Hojati, H. Touati, R. Kurshan, and R. Brayton. Efficient ω-regular language containment. *International Conference on Computer-Aided Verification*, 1992.
10. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata, Languages and Computation.* Addison-Wesley, 1979.
11. H. Touati, R. Brayton, and R. Kurshan. Testing language containment for ω-automata using bdd's. *International Workshop on Formal Methods in VLSI Design*, 1991.