Efficient Verification of Real-Time Systems: Compact Data Structure and State–Space Reduction

Kim G. Larsen* Fredrik Larsson† Paul Pettersson† Wang Yi†

Abstract

During the past few years, a number of verification tools have been developed for real-time systems in the framework of timed automata (e.g. KRONOS and UPPAAL). One of the major problems in applying these tools to industrial-size systems is the huge memory-usage for the exploration of the state-space of a network (or product) of timed automata, as the model-checkers must keep information on not only the control structure of the automata but also the clock values specified by clock constraints.

In this paper, we present a compact data structure for representing clock constraints. The data structure is based on an $\mathcal{O}(n^3)$ algorithm which, given a constraint system over real-valued variables consisting of bounds on differences, constructs an equivalent system with a *minimal* number of constraints. In addition, we have developed an on-the-fly reduction technique to minimize the space-usage. Based on static analysis of the control structure of a network of timed automata, we are able to compute a set of symbolic states that cover all the dynamic loops of the network in an on-the-fly searching algorithm, and thus ensure termination in reachability analysis.

The two techniques and their combination have been implemented in the tool UPPAAL. Our experimental results demonstrate that the techniques result in truly significant space-reductions: for six examples from the literature, the space saving is between 75% and 94%, and in (nearly) all examples time-performance is improved. Also noteworthy is the observation that the two techniques are completely orthogonal.

1 Introduction

Reachability analysis has been one of the most successful methods for automated analysis of concurrent systems. Many verification problems e.g. trace–inclusion and invariant checking can be solved by means of reachability analysis. It can in many cases also be used for checking whether a system described as an automaton satisfies a requirement specification formulated e.g. in linear temporal logic, by converting the requirement to an automaton and thereafter checking whether the parallel composition of the system and requirement automata can reach certain annotated states [31, 20]. However, the major problem in applying reachability analysis is the potential combinatorial explosion of state spaces. To attack this problem, various symbolic and reduction techniques have been put forward over the last decade to efficiently represent state space and to avoid exhaustive state space exploration (e.g. [10, 16, 30, 11, 12, 15, 4]); such techniques have played a crucial role for the successful development of verification tools for finite–state systems.

In the last few years, new verification tools have been developed, for the class of infinite–state systems known as timed systems [18, 13, 8]. Notably the verification engines of most tools in this category are based on reachability analysis on timed automata following the pioneering work of Alur and Dill [3]. A timed automaton is an extension of a finite automaton with a finite set

*BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation), Department of Computer Science and Mathematics, Aalborg University, Denmark. E-mail: kgl@cs.auc.dk.

†Department of Computer Systems, Uppsala University, Sweden. E-mail: {fredrikl,paupet,yi}@docs.uu.se.

PASSED:= $\{\}$
WAITING:= $\{(l_0, D_0)\}$
repeat
  begin
    get (l, D) from WAITING
    if $(l, D) \models \varphi$ **then return "YES"**
    else if $D \not\subseteq D'$ **for all** $(l, D') \in$ PASSED **then**
      begin
        add (l, D) to PASSED
        SUCC:=$\{(l_s, D_s) : (l, D) \rightsquigarrow (l_s, D_s) \wedge D_s \neq \emptyset\}$
        for all $(l_{s'}, D_{s'})$ in SUCC **do**
          put $(l_{s'}, D_{s'})$ to WAITING
      end
  end
until WAITING=$\{\}$
return "NO"

Figure 1: An Algorithm for Symbolic Reachability Analysis.

of real-valued clock-variables. The foundation for decidability of reachability problems for timed automata is Alur and Dill's region technique, by which the infinite state space of a timed automaton due to the density of time, may effectively be partitioned into finitely many equivalence classes i.e. *regions* in such a way that states within each class will always evolve to states within the same classes. However, reachability analysis based on the region technique is practically infeasible due to the potential state explosions arising from not only the control-structure (as for finite–state systems) but also the region space [22].

Efficient data structures and algorithms have been sought to represent and manipulate timing constraints over clock variables (e.g. by Difference Bounded Matrices [6, 14, 32], or Binary Decision Diagrams [10, 5]) and to avoid exhaustive state space exploration (e.g. by application of partial order reductions [16, 30, 25] or compositional methods [4, 22]). One of the main achievements in these studies is the symbolic technique [14, 32, 19, 33, 22], that converts the reachability problem to that of solving simple constraints systems. The technique can be simply formulated in an abstract reachability algorithm[1] as shown in Figure 1. The algorithm is to check whether a timed automaton may reach a state satisfying a given state formula φ. It explores the state space of the automaton in terms of *symbolic states* in the form (l, D) where l is a control–node and D is a constraint system over clocks variables.

We observe that several operations of the algorithm are critical for efficient implementations. Firstly, the algorithm depends heavily on the test operations for checking the inclusion $D \subseteq D'$ (i.e. the inclusion between the solution sets of D, D') and the emptiness of D_s in constructing the successor set SUCC of (l, D) . Clearly, it is important to design efficient data structures and algorithms for the representation and manipulation of clock constraints. One such well–known data structure is that of DBM (*Difference Bounded Matrix*), which offers a canonical representation for constraint systems. It has been successfully employed by several real–time verification tools, e.g. UPPAAL [8] and KRONOS [13]. A DBM representation is in fact a weighted directed graph where the vertices correspond to clocks (including a zero-clock) and the weights on the edges stand for the bounds on the differences between pairs of clocks [6, 14, 32]. As it gives an explicit bound for the difference between each pair of clocks, its space–usage is in the order of $\mathcal{O}(n^2)$ where n is the number of clocks. However, in practice it often turns out that most of these bounds are redundant.

In this paper, we present a compact data structure for DBM, which provides *minimal* and *canonical* representations of clock constraints and also allows for efficient inclusion checks. We

[1] Several verification tools for timed systems (e.g. UPPAAL [8]) have been implemented based on this algorithm.

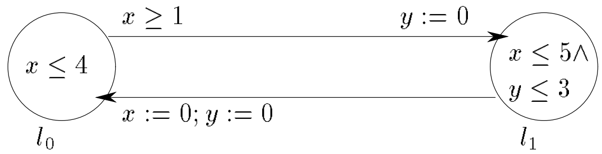

Figure 2: A Timed Automaton.

have developed an $\mathcal{O}(n^3)$ algorithm that given a DBM constructs a minimal number of constraints equivalent to the original constraints represented by the DBM (i.e. with the same solution set). The algorithm is essentially a minimization algorithm for weighted directed graphs, and hence solves a problem of independent interest. Note that the main global datastructure of the algorithm in Figure 1 is the passed list (i.e. PASSED) holding the explored states. In many cases, it will store all the reachable symbolic states of the automaton. Thus, it is desirable that when saving a (symbolic) state in the passed list, we save the (often substantially smaller) minimal constraint system. Also, the minimal representation makes the inclusion–checking of the algorithm more efficient. Our experimental results demonstrate truly significant space–savings as well as better time–performance (see statistics in section 5).

In addition to the *local* reduction technique above, which is to minimize the space–usage of each individual symbolic state, as the second contribution of this paper, we have developed a *global* reduction technique to reduce the total number of states to save in the global datastructure, i.e. the passed list. It is completely orthogonal to the local technique. In the abstract algorithm of Figure 1, we notice the step of saving the new encountered state (l, D) in the passed list when the inclusion–checking for $D \subseteq D'$ fails (i.e. $D \not\subseteq D'$). Its purpose is first of all to guarantee termination but also to avoid repeated exploration of states that have several predecessors. However, this is not necessary if all the predecessors of (l, D) are already present in the passed list. In fact, to ensure termination, it suffices to save only one state for each dynamic loop. An improved on–the–fly reachability algorithm accoding to the global reduction strategy has been implemented in UPPAAL [8] based on statical analysis of the control structure of timed automata. Our experimental results demonstrate significant space–savings and also better time–performance (see statistics in section 5).

The outline of this paper is as follows: In the next section we review the semantics of timed automata and the notion of Difference Bounded Matrix (DBM) for clock constraints. Section 3 presents the compact datastructure for DBM and the local reduction technique (i.e. the minimization algorithm for weigthed directed graphs). Section 4 is devoted to develop the global reduction technique based on control structure analysis. Section 5 presents our experimental results for both techniques and their combination. Section 6 concludes the paper.

2 Preliminaries

2.1 Timed Automata

Timed automata was first introduced in [3] and has since then established itself as a standard model for real–time systems. For the reader not familiar with the notion of timed automata we give a short informal description.

Consider the timed automaton of Figure 2. It has two control nodes l_0 and l_1 and two real–valued clocks x and y. A *state* of the automaton is of the form (l, s, t), where l is a control node, and s and t are non–negative reals giving the value of the two clocks x and y. A control node is labelled with a condition (the invariant) on the clock values that must be satisfied for states involving this node. Assuming that the automaton starts to operate in the state $(l_0, 0, 0)$, it may stay in node l_0 as long as the invariant $x \leq 4$ of l_0 is satisfied. During this time the values of the clocks increase synchronously. Thus from the initial state, all states of the form (l_0, t, t), where

$t \leq 4$, are reachable. The edges of a timed automaton may be decorated with a condition (guard) on the clock values that must be satisfied in order to be enabled. Thus, only for the states (l_0, t, t), where $1 \leq t \leq 4$, is the edge from l_0 to l_1 enabled. Additionally, edges may be labelled with simple assignments reseting clocks. E.g. when following the edge from l_0 to l_1 the clock y is reset to 0 leading to states of the form $(l_1, t, 0)$, where $1 \leq t \leq 4$.

Thus, a timed automaton is a standard finite–state automaton extended with a finite collection of real–valued clocks ranged over by x, y etc. We use $\mathcal{B}(C)$ ranged over by g (and latter D), to stand for the set of formulas that can be an atomic constraint of the form: $x \sim n$ or $x - y \sim n$ for $x, y \in C$, $\sim \in \{\leq, \geq\}$[2] and n being a natural number, or a conjunction of such formulas. Elements of $\mathcal{B}(C)$ are called *clock constraints* or *clock constraint systems* over C.

Definition 1 *A timed automaton A over clocks C is a tuple $\langle N, l_0, E, I\rangle$ where N is a finite set of nodes (control-nodes), l_0 is the initial node, $E \subseteq N \times \mathcal{B}(C) \times 2^C \times N$ corresponds to the set of edges, and finally, $I : N \rightarrow \mathcal{B}(C)$ assigns invariants to nodes. In the case, $\langle l, g, r, l'\rangle \in E$, we write $l \xrightarrow{g,r} l'$.* □

Formally, we represent the values of clocks as functions (called clock assignments) from C to the non–negative reals $\mathbf{R}$. We denote by $\mathbf{R}^C$ the set of clock assignments for C. A semantical *state* of an automaton A is now a pair (l, u), where l is a node of A and u is a clock assignment for C, and the semantics of A is given by a transition system with the following two types of transitions (corresponding to delay–transitions and edge–transitions):

- $(l, u) \rightarrow (l, u + d)$ if $I(u)$ and $I(u + d)$
- $(l, u) \rightarrow (l', u')$ if there exist g and r such that $l \xrightarrow{g,r} l'$, $u \in g$ and $u' = [r \rightarrow 0]u$

where for $d \in \mathbf{R}$, $u + d$ denotes the time assignment which maps each clock x in C to the value $u(x) + d$, and for $r \subseteq C$, $[r \mapsto 0]u$ denotes the assignment for C which maps each clock in r to the value 0 and agrees with u over $C \backslash r$. By $u \in g$ we denote that the clock assignment u satisfies the constraint g (in the obvious manner).

Clearly, the semantics of a timed automaton yields an infinite transition system, and is thus not an appropriate basis for decision algorithms. However, efficient algorithms may be obtained using a finite–state *symbolic* semantics based on *symbolic states* of the form (l, D), where $D \in \mathcal{B}(C)$ [19, 33]. The symbolic counterpart to the standard semantics is given by the following two (fairly obvious) types of symbolic transitions:

- $(l, D) \rightsquigarrow \Big(l, (D \wedge I(l))^{\uparrow} \wedge I(l)\Big)$
- $(l, D) \rightsquigarrow \Big(l', r(g \wedge D)\Big)$ if $l \xrightarrow{g,r} l'$

where $D^{\uparrow} = \{u + d \mid u \in D \wedge d \in \mathbf{R}\}$ and $r(D) = \{[r \rightarrow 0]u \mid u \in D\}$. It may be shown that $\mathcal{B}(C)$ (the set of constraint systems) is closed under these two operations ensuring the well–definedness of the semantics. Moreover, the symbolic semantics corresponds closely to the standard semantics in the sense that, whenever $u \in D$ and $(l, D) \rightsquigarrow (l', D')$ then $(l, u) \rightarrow (l', u')$ for some $u' \in D'$.

Finally, we introduce the notion of networks of timed automata [33, 22]. A network is the parallel composition of a finite set of automata for a given synchronization function. However, to illustrate the on–the–fly verification technique, we only need to study the case dealing with interleaving, that is, the network of automata $A_1 \ldots A_n$, is the cartesian product of A_i's. Assume a vector $\bar{l}$ of control nodes. We shall use $\bar{l}[i]$ to stand for the ith element of $\bar{l}$ and $\bar{l}[l'_i/l_i]$ for the vector where the ith element l_i of $\bar{l}$ is replaced by l'_i. A control node (i.e. *control vector*) $\bar{l}$ of a network A_i's is a vector where $\bar{l}[i]$ is a node of A_i and the invariant $I(\bar{l})$ of $\bar{l}$ is the conjuction of $I(\bar{l}[1]) \ldots I(\bar{l}[n])$. The symbolic semantics of networks is given in terms of control vectors [22].

[2]For reasons of simplicity and clarity in presentation we have chosen only to consider the non–strict orderings. However, the techniques given extends easily to strict orderings.

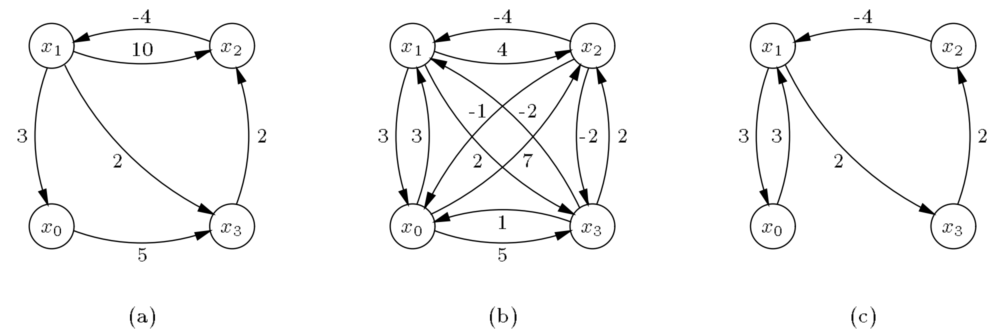

Figure 3: Graph for E (a), its shortest–path closure (b), and shortest–path reduction (c).

- $(\bar{l}, D) \rightsquigarrow \left(\bar{l}, (D \wedge I(\bar{l}))^{\uparrow} \wedge I(\bar{l})\right)$
- $(\bar{l}, D) \rightsquigarrow \left(\bar{l}[l_i'/l_i], r(g \wedge D)\right)$ if $l_i \xrightarrow{g,r} l_i'$

In the later case, we shall say that the symbolic transition is derived by the edge $l_i \xrightarrow{g,r} l_i'$.

2.2 Difference Bounded Matrices & Shortest–Path Closure

To utilize the above symbolic semantics algorithmically, as for example in the reachability algorithm of Figure 1, it is important to design efficient data structures and algorithms for the representation and manipulation of clock constraints.

One such well–known data structure is that of difference bounded matrices (DBM, see [6, 14]), which offers a canonical representation for constraint systems. A DBM representation of a constraint system D is simply a weighted, directed graph, where the vertices correspond to the clocks of C and an additional zero–vertex 0. The graph has an edge from x to y with weight m provided $x - y \leq m$ is a constraint of D. Similarly, there is an edge from 0 to x (from x to 0) with weight m, whenever $x \leq m$ ($x \geq -m$) is a constraint of D[3]. As an example, consider the constraint system E over $\{x_0, x_1, x_2, x_3\}$ being a conjunction of the atomic constraints $x_0 - x_1 \leq 3$, $x_3 - x_0 \leq 5$, $x_3 - x_1 \leq 2$, $x_2 - x_3 \leq 2$, $x_2 - x_1 \leq 10$, and $x_1 - x_2 \leq -4$. The graph representing E is given in Figure 3 (a).

In general, the same set of clock assignments may be described by several constraint systems (and hence graphs). To test for inclusion between constraint systems D and D' [4], which we recall is essential for the termination of the reachability algorithm of Figure 1, it is advantageous, that D is *closed under entailment* in the sense that no constraint of D can be strengthened without reducing the solution set. In particular, for D a closed constraint system, $D \subseteq D'$ holds if and only if for any constraint in D' there is a constraint in D at least as tight; i.e. whenever $(x - y \leq m) \in D'$ then $(x - y \leq m') \in D$ for some $m' \leq m$. Thus, closedness provides a canonical representation, as two closed constraint systems describe the same solution set precisely when they are identical. To close a constraint system D simply amounts to derive the shortest–path closure for its graph and can thus be computed in time $\mathcal{O}(n^3)$, where n is the number of clocks of D . The graph representation of the closure of the constraint system E from Figure 3 (a) is given in Figure 3 (b). The emptiness-check of a constraint system D simply amounts to checking for negative–weight cycles in its graph representation. Finally, given a closed constraint system D the operations $D^{\uparrow}$ and $r(D)$ may be performed in time $\mathcal{O}(n)$.

[3] We assume that D has been simplified to contain at most one upper and lower bound for each clock and clock–difference.

[4] To be precise, it is the inclusion between the *solution sets* for D and D'.

3 Minimal Constraint Systems & Shortest Path Reductions

For the reasons stated above a matrix representation of constraint systems in closed form is an attractive data structure, which has been successfully employed by a number of real–time verification tools, e.g. UPPAAL [8] and KRONOS [13]. As it gives an explicit (tightest) bound for the difference between each pair of clocks (and each individual clock), its space–usage is of the order $\mathcal{O}(n^2)$. However, in practice it often turns out that most of these bounds are redundant, and the reachability algorithm of Figure 1 is consequently hampered in two ways by this representation. Firstly, the main–data structure PASSED, will in many cases store all the reachable symbolic states of the automaton. Thus, it is desirable, that when saving a symbolic state in the PASSED-list, we save a representation of the constraint–system with as few constraints as possible. Secondly, a constraint system D added to the PASSED-list is subsequently only used in checking inclusions of the form $D' \subseteq D$. Recalling the method for inclusion–check from the previous section, we note that (given D' is closed) the time–complexity of the inclusion–check is linear in the number of constraints of D. Thus, again it is advantageous for D to have as few constraints as possible.

In the following subsections we shall present an $\mathcal{O}(n^3)$ algorithm, which given a constraint system constructs an equivalent reduced system with the minimal number of constraints. The reduced constraint system is canonical in the sense that two constrain systems with the same solution set give rise to identical reduced systems. The algorithm is essentially a minimization algorithm for weighted directed graphs. Given a weighted, directed graph with n vertices, it constructs in time $\mathcal{O}(n^3)$ a reduced graph with the minimal number of edges having the same shortest path closure as the original graph. Figure 3 (c) shows the minimal graph of the graphs in Figure 3 (a) and (b), which is computed by the algorithm.

3.1 Reduction of Zero–Cycle Free Graphs

A weighted, directed graph G is a structure (V, E_G), where V is a finite set of vertices and E_G, is a partial function from $V \times V$ to Z (the integers). The domain of E_G constitutes the edges of G, and when defined, $E_G(x,y)$ gives the weight of the edge between x and y. We assume that $E_G(x,x) = 0$ for all vertices x, and that G has no cycles with negative weight[5].

Given a graph G, we denote by G^C the *shortest–path closure* of G, i.e. $E_{G^C}(x,y)$ is the length of the shortest path from x to y in G. A *shortest–path reduction* of a graph G is a graph G^R with the minimal number of edges such that $\left(G^R\right)^C = G^C$.

The key to reduce a graph is obviously to remove *redundant edges*, where an edge (x,y) is redundant if there exist an alternative path from x to y whose (accumulated) weight does not exceed the weight of the edge itself. E.g. in the graph of Figure 3 (a), the edge (x_1, x_2) is clearly redundant as the accumulated weight of path $(x_1, x_0), (x_0, x_3), (x_3, x_2)$ has a weight (10) not exceeding the weight of the edge itself (also 10). Also, the path $(x_1, x_3), (x_3, x_2)$ makes the edge (x_1, x_2) redundant. Being redundant, the edge (x_1, x_2) may be removed without changing the shortest–path closure.

Now, consider the edge (x_1, x_2) in the graph of Figure 3 (b). Clearly, the edge is redundant as the path $(x_1, x_3), (x_3, x_2)$ has equal weight. Similarly, the edge (x_3, x_2) is redundant as the path $(x_3, x_1), (x_1, x_2)$ has equal weight. However, though redundant, we cannot just remove the two edges (x_1, x_2) and (x_3, x_2) as removal of one clearly requires the presence of the other. In fact, all edges between the vertices x_1, x_2 and x_3 are redundant, but obviously we cannot remove them all simultaneously. The key explanation of this complicating phenomena is that x_1, x_2, x_3 constitutes a cycle with length zero (a *zero–cycle*). However, for zero–cycle free graphs the situation is the simplest possible:

Lemma 1 *Let G_1 and G_2 be zero–cycle free graphs such that ${G_1}^C = {G_2}^C$. If there is an edge $(x,y) \in G_1$ such that $(x,y) \notin G_2$, then $\left(G_1 \backslash \{(x,y)\}\right)^C = {G_1}^C = {G_2}^C$.* □

Proof See Appendix A.

[5] This would correspond to constraint systems with empty solution set.

From the above Lemma it follows immediately that all redundant edges of a zero–cycle free graph may be removed without affecting the closure. On the other hand, removal of an edge which is not redundant will of course change the closure of the graph, and must be present in any graph with the same closure. Thus the following Theorem follows:

Theorem 1 *Let G be a zero–cycle free graph, and let $\{\alpha_1, \dots, \alpha_2\}$ be the set of redundant edges of G. Then $G^R := G^C \backslash \{\alpha_1, \dots, \alpha_k\}$.* □

From an algorithmic point of view, redundancy of edges is easily determined given the closure G^C of a graph G as only path of length 2 needs to be considered: an edge (x, y) is redundant precisely when there is a vertex z $(\neq x, y)$ such that $E_{G^C}(x, y) \geq E_{G^C}(x, z) + E_{G^C}(z, y)$. Thus for zero–cycle free graphs the reduction can clearly be computed in time $\mathcal{O}(n^3)$.

3.2 Reduction of Negative–Cycle Free Graphs

For general graphs (without negative cycles) our reduction construct relies on a partitioning of the vertices according to zero–cycles. We say that two vertices x and y are *equivalent* or *zero–equivalent*, if there is a zero–cycle containing them both. We write $x \equiv y$ in this case. Given the closure G^C of a graph G, it is extremely easy to check for zero–equivalence: $x \equiv y$ holds precisely when $E_{G^C}(x, y) = -E_{G^C}(y, x)$. Thus, in the graphs of Figure 3 (a) and (b), $\equiv$ partitions the vertices into the two classes $\{x_0\}$ and $\{x_1, x_2, x_3\}$.

To obtain a canonical reduction, we assume that the vertices of G are ordered by assigning them indices as $x_1, x_2, \dots, x_n$. The equivalence $\equiv$ now induces a natural transformation on the graph G:

Definition 2 *Given a graph G, the vertices of the graph $G_\equiv$ are vertex equivalence classes, denoted E_k, of G with respect to $\equiv$. There is an edge between the classes E_i and E_j $(i \neq j)$ if for some $x \in E_i$ and $y \in E_j$ there is an edge in G between x and y. The weight of this edge is $E_{G^C}(\min E_i, \min E_j)$, where $\min E$ is the vertex in E with the smallest index.* □

Thus, the distance between E_i and E_j in $G_\equiv$ is the weight of the shortest path in G between the elements of E_i and E_j with smallest index. It is obvious that $G_\equiv$ is a zero–cycle free graph. Also, it is easy to see that $G_{1\equiv} = G_{2\equiv}$ if ${G_1}^C = {G_2}^C$. Let H be the graph of Figure 3 (a). Then $H_\equiv$ will have vertices $E_0 = \{x_0\}$ and $E_1 = \{x_1, x_2, x_3\}$. The two vertices are connected by two edges both having weight 3.

The following provides a dual to the operator of Definition 2:

Definition 3 *Let F be a graph with vertices being $\equiv$–equivalence classes with respect to a graph $G = (V, E_G)$. Then the* expansion *of F is a graph F^+ with vertices V and with weight satisfying:*

- *For any multi-member equivalence class $\{z_1 < z_2 < \cdots < z_k\}$[6] of F, F^+ contains a single cycle $z_1, z_2, \dots, z_k, z_1$, with the weight of the edge (z_i, z_{i+1}) being the weight of the shortest path from z_i to z_{i+1} in G.*
- *Whenever (E_i, E_j) is an edge in F with weight m, then F^+ will have an edge from $\min E_i$ to $\min E_j$ with weight m.* □

We are now ready to state the Main Theorem giving the shortest–path reduction construct for arbitrary negative–cycle free graphs:

Theorem 2 *Let G be negative–cycle free graph. Then the shortest–path reduction of G is given by the graph:*

$$G^R := \left({G_\equiv}^R\right)^+$$

□

[6] $<$ refers to the assumed ordering on the vertices of G.

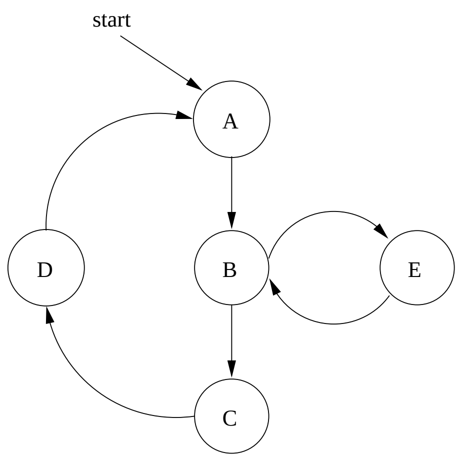

Figure 4: Illustration of Space–Reduction.

Proof See Appendix A.

First, note that the above definition is well–defined as $G_{\equiv}$ is a zero–cycle free graph and the reduction construction of Theorem 1 thus applies. Given the closure G^C of G the constructions of Definitions 2 and 3 are easily seen to be computable in $\mathcal{O}(n^2)$; it follows that G^R can be constructed in $\mathcal{O}(n^3)$. Now applying the above construction to the graph H of Figure 3 (a), we first note that ${H_{\equiv}}^R = H_{\equiv}$ as $H_{\equiv}$ has no redundant edges. Now expanding $H_{\equiv}$ with respect to the vertex ordering $x_0 < x_1 < x_2 < x_3$ gives the graph of Figure 3 (c), which according to Theorem 2 above is the shortest–path reduction of H.

Experimental results show that the use of minimal constrain systems (obtained by the above shortest–path reduction algorithm) as a compact data structure leads to truly significant space–savings in practical reachability analysis of timed systems: the space–savings are in the range 70–86%. We refer to section 5 for more details.

4 Global Reductions and Control Structure Analysis

The preceding section is about *local* reductions in reachability analysis in the sense that the technique developed is for *each* individual symbolic state. In this section, we shall develop a *global* reduction technique to reduce the total number of symbolic states to save in the *global* data structure i.e. the passed list.

4.1 Potential Space–Reductions

We recall the standard reachability analysis algorithm for finite graphs (see e.g. [26]). It is similar to the one in Figure 1, but simpler as no constraints but only control nodes are involved. The algorithm repeats three main operations: *examining* every new encountered node (to see if it is in the passed list), *exploring* the new encountered nodes (computing all their successors for further analysis) and *saving* the explored nodes in the passed list until all reachable nodes are present in the list (that is, all new encountered nodes are already in the passed list).

Note that the saving of an explored node is to ensure termination and also to avoid repeated exploration of nodes with more than one incoming edges. However it is not necessary to save all reachable nodes. Consider for example, the simple graph in Figure 4. Clearly, there is no need to save node C, D or E as they will be visited only once if B is present in the passed list.

In fact, to guarantee termination on a finite graph, it is sufficient to save only one node for each cycle in the graph. For example, as B covers the two cycles of the graph in Figure 4, in addition to C, D, and E, it is not necessary to save A either. In general, *for a finite graph, there is a minimal number of nodes to save in the passed list in order to guarantee termination*. However the trade–off of the space-saving strategy may be increased time–consumption. Consider the same graph of Figure 4. If node A is not present in the passed list, it will be explored again whenever D is explored. This can be avoided by saving A when it is first visited. But the difference from saving B is that saving A is for efficiency and B for termination. More generally, the following is true of finite graphs:

Proposition 1 *For a finite graph, there is a minimal number of nodes to save in order to guarantee termination as well as that every reachable node will be explored*[7] *only once in reachability analysis.* □

Now we recall the abstract reachability algorithm in Figure 1 for timed systems. To ensure termination and also to avoid repeated exploration of states (that have more than one predecessors), it saves every new encountered state (l, D) in the passed list when the inclusion–checking for $D \subseteq D'$ fails (i.e. $D \not\subseteq D'$). Obviously this is not necessary if all the predecessors of (l, D) already exist in the PASSED–list. Similar to the case for finite graphs, for termination, we need to save only one state for every *dynamic loop* of a timed automaton.

Definition 4 *(Dynamic Loops) A set L_d of symbolic states $(l_1, D_1)\dots(l_n, D_n)$ of a timed automaton is a dynamic loop if $(l_1, D_1) \rightsquigarrow (l_2, D_2)\dots(l_{n-1}, D_{n-1}) \rightsquigarrow (l_n, D_n)$ and $(l_n, D_n) \rightsquigarrow (l_1, D_1')$ with $D_1' \subseteq D_1$. A symbolic state is said to cover a dynamic loop if it is a member of the loop.* □

We claim that to ensure termination, it is sufficient (but not necessary) to save a set of symbolic states that cover all the dynamic loops. Now, the problem is how to compute efficiently such a set.

4.2 Control Structure Analysis and Application

We shall utilize the statical structure of an automaton to identify potential candidates of states to cover dynamic loops.

Definition 5 *(Statical Loops and Entry Nodes) A set L of nodes $l_1, ..., l_n$ of a timed automaton is a statical loop if there is a sequence of edges $l_1 \to l_2 \dots l_{n-1} \to l_n$ and $l_n \to l_1$ where $l_i \to l_j$ denotes that $l_i \xrightarrow{g,r} l_j$ for some g, r is an edge of the automaton. A node l_i of a statical loop L is an entry node of L if it is an initial node or there exists a node $l \notin L$ (outside of the loop) and an edge $l \to l_i$. Further, we say that a vector of nodes (i.e. a node of a network) is an entry node if any of its components is.* □

For example, nodes A, B, C and D in Figure 4 constitute a statical loop with entry nodes A and B; another statical loop is nodes B and E with entry node B. In general, since the sets of control nodes and edges of a timed automaton are finite, the number of statical loops is finite and so is the set of entry nodes of all statical loops. In fact the set of entry nodes of a timed automaton can be easily computed by statical analysis using a stack or a slightly modified loop detecting algorithm (e.g. [28]).

Now note that according to Definition 4, a dynamic loop (a set of symbolic states) must contain a subset of symbolic states whose control nodes constitute a statical loop. As a statical loop always contains an entry node, we have the following fact.

Proposition 2 *Every dynamic loop of a timed automaton contains at least one symbolic state (l, D) where l is an entry node.* □

Following Proposition 2, to cover all the dynamic loops, we may simply save all the states whose control–nodes are an entry node, and ignore the others. Obviously, this will not give much reduction when dynamic loops include mostly entry nodes, which is the case when a network of automata contains a component whose nodes are mostly entry nodes e.g. a testing automaton. For networks of automata, we adopt the strategy of saving the *first derived* states whose control nodes are an entry node, known as *covering states* in the following sense.

[7] Note that a state is explored means that all its successors are generated for further analysis.

	Current		CDSC		CSR		CDSC & CSR	
	space	time	space	time	space	time	space	time
Audio	828	0.44	219	0.43	774	0.44	206	0.47
Audio w. Coll.	646 092	3 465.22	198 178	2 067.37	370 800	1 515.88	111 632	929.22
B & O	778 288	13 240.49	249 175	6 967.38	642 752	9 348.48	204 795	4 966.79
Box Sorter	625	0.20	139	0.18	175	0.41	36	0.41
M. Plant	92 592	155.61	27 042	39.85	50 904	56.61	14 933	24.22
Mutex 2	225	0.13	44	0.14	99	0.15	18	0.14
Mutex 3	3 376	1.40	621	0.67	1 360	0.65	240	0.51
Mutex 4	56 825	102.49	9 352	24.48	22 125	25.97	3 532	12.14
Mutex 5	1 082 916	14 790.56	158 875	3 299.96	416 556	3 111.21	59 720	1 138.32
Train Crossing	464	0.19	130	0.18	384	0.20	114	0.18

Table 1: Performance Statistics.

Definition 6 *Assume a network of timed automata with an initial state* $(\bar{l}_0, D_0)$ *and a given symbolic state* $(\bar{l}, D)$. *We say that* $(\bar{l}, D)$ *is a covering state of the network if it is reachable in the sense that there exists a sequence of symbolic transitions* $(\bar{l}_0, D_0) \rightsquigarrow (\bar{l}_1, D_1) \ldots (\bar{l}_n, D_n) \rightsquigarrow (\bar{l}, D)$ *and an* i *(standing for the ith component of the network) such that* $\bar{l}[i]$ *is an entry node and* $(\bar{l}_n, D_n) \rightsquigarrow (\bar{l}, D)$ *is derived by an edge* $\bar{l}_n[i] \xrightarrow{g,r} \bar{l}[i]$ *for some* g *and* r. □

From the above definition, it should be obvious that we can easily decide whether a reachable symbolic state is a covering state by an on–the–fly algorithm when the entry nodes of all the component automata are known through statical analysis as discussed earlier.

Finally, we claim that the set of *covering* states of a network covers all its dynamic loops and therefore it suffices to keep them in the passed list for the sake of termination in reachability analysis [8].

Theorem 3 *Every dynamic loop of a network of timed automata contains at least one covering state.* □

Proof See Appendix A.

An improved reachability algorithm according to the saving strategy induced from Theorem 3 (i.e. saving only the covering sates in the passed list) has been implemented in UPPAAL. Our experimental results show significant space–reductions and also better time–performance (see Table 1 in section 5).

5 Experimental Results

The techniques developed in preceding sections have been implemented and added to the tool UPPAAL [8]. In this section we present the results of an experiment where both the original version of UPPAAL and its extensions were applied to verify six well-studied examples from the literature.

Philips Audio Protocol (Audio) The protocol was developed and implemented by Philips to exchange control information between components in audio equipment using Manchester encoding. The correctness of the encoding relies on timing delays between signals. It is firstly studied and manually verified in [9].

We have verified using UPPAAL that the main correctness property holds of the protocol, i.e. all bit streams sent by the sender are correctly decoded by the receiver [23], if the timing error is ±5%.

[8] Note that this is only a sufficient condition but not necessary.

Philips Audio Protocol with Bus Collision (Audio w. Coll.) This is an extended variant of Philips audio control protocol with bus collision detection [7]. It is significantly larger than the version above since several new components (and variables) are introduced, and existing components are modified to deal with bus collisions.

In this experiment we verified that correct bit sequences are received by the receiver (i.e. Property 1 of [7]), using the error tolerances set by Philips.

Bang & Olufsen Audio/Video Protocol (B&O) This is an audio control protocol highly dependent on real-time. The protocol is developed by Bang & Olufsen, to transmit messages between audio/video components over a single bus, and further studied in [17]. Though it was known to be faulty, the error was not found using conventional testing methods. Using UPPAAL, an error-trace is automatically produced, which reviled the error. Furthermore, a correction is suggested and automatically proved using UPPAAL.

In this experiment we have verified that the protocol is correct.

Box Sorter (Box Sorter) The example of [24] is a model of a sorter unit that sorts red and blue boxes. When the boxes moves down a lane they pass a censor and a piston. The sorter reads the information from the censor and sorts out the red boxes by controlling the position of the piston. We have verified that the sorter is correct.

Manufacturing Plant (M. Plant) The example is a model of the manufacturing plant of [27, 13]. It is a production cell with: a 50 feets belt moving from left to right, two boxes, two robots and a service station. Robot A moves boxes off the rightmost extreme of the belt to the service station. Robot B moves boxes from the service station to the left-most extreme of the belt.

Assuming an initial distance between the boxes on the belt we verified that no box will fall off the belt.

Mutual Exclusion Protocol (Mutex 2 – Mutex 5) It is the so-called Fischers protocol that has been studied previously in many experiments, e.g. [1, 29]. The protocol is to ensure mutual exclusion among several processes competing for a critical section using timing constraints and a shared variable. In the experiment we use the full version of the protocol where a process may recover from failed attempts to enter the critical section, and also eventually leave the critical section [21].

The protocol is verified to enjoy the invariant property: there is never more than one process existing in the critical section. The results for 2 to 5 processes are shown in Table 1.

Train Crossing Controller (Train Crossing) It is a variant of the train gate controller [18]. An approaching train signals to the controller which reacts by closing the gate. When the train have passed the controller opens the crossing. We have verified that the gate is closed whenever a train is close to the crossing.

In Table 1 we present the space (in number of timing constraints stored on the PASSED-buffer) and time requirements (in seconds) of the examples on a Sun SPARCstation4 equipped with 64 MB of primary memory. Each example was verified using the current algorithm of UPPAAL (Current), and using modified algorithms for: Compact Data Structures for Constraints (CDSC), Control Structure Reduction (CSR), and their combination (CDSC & CSR).

As shown in Table 1 both techniques give truly significant space savings: CDSC saves 68–85% of the original consumed space, CSR demonstrates more variation saving 13–72%, and both methods (nearly) uniformly result in better time-performance. Most significant is that the two techniques are completely orthogonal, witnessed by the numbers for the combined technique which shows a space-saving between 75% and 94%.

6 Conclusion

In this paper, we have two contributions to the development of efficient data structure and algorithms for the automated analysis of timed systems.

Firstly, we have presented a compact data structure, for representing the subsets of Euclidean space that arise during verification of timed automata, which provides *minimal* and *canonical* representations for clock constraints, and also allows for efficient inclusion checks between constraint systems. The data structure is based on an $\mathcal{O}(n^3)$ algorithm which, given a constraint systems over real–valued variables consisting of bounds on differences, constructs an equivalent system with a minimal number of constraints. It is essentially a minimization algorithm for weighted directed graphs, that extends the transitive reduction algorithm of [2] to weighted graphs. Given a weighted, directed graph with n vertices, it constructs in time $\mathcal{O}(n^3)$ a reduced graph with the minimal number of edges having the same shortest path closure as the original graph.

In addition, we have developed an on–the–fly reduction technique to minimize the space–usage by reducing the total number of symbolic states to save in reachability analysis for timed systems. The technique is based on the observation that to ensure termination in reachability analysis, it is not necessary to save all the explored states in memory, but certain critical states. Based on static analysis of the control structure of timed automata, we are able to compute a set of covering states that cover all the dynamic loops of a system. The covering set may not be minimal but sufficient to guarantee termination in an on–the–fly reachability algorithm.

The two techniques and their combination have been implemented in the tool UPPAAL. Our experimental results demonstrate that the techniques result in truly significant space–reductions: for a number of well–studied examples in the literature the space saving is between 74% and 97%, and in (nearly) all examples time–performance is improved. Also noteworthy is the observation that the two techniques are completely orthogonal.

As future work, we wish to further study the global on–the–fly reduction technique to identify the *minimal* sets of covering states that ensure termination and also avoid repeated explorations in reachability analysis for timed systems.

References

[1] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, 1993.

[2] A. Aho, M. Garey, and J. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal of Computing*, 1(2):131–137, June 1972.

[3] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, 1990.

[4] H. R. Andersen. Partial Model Checking. In *Proc. of Symp. on Logic in Computer Science*, 1995.

[5] E. Asarin, O. Maler, and A. Pnueli. Data–structures for the verification of timed automata. Accepted for presentation at HART97, 1997.

[6] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[7] J. Bengtsson, D. Griffioen, K. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In R. Alur and T. A. Henzinger, editors, *Proc. of 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Proc. of the 2nd Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, Mars 1996.

[9] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, 1994.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. *Logic in Computer Science*, 1990.

[11] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.

[12] E. M. Clarke, O. Grümberg, and D. E. Long. Model Checking and Abstraction. *Principles of Programming Languages*, 1992.

[13] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 66–75, Dec. 1995.
[14] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 197–212. Springer–Verlag, 1989.
[15] E. A. Emerson and C. S. Jutla. Symmetry and Model Checking. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, 1993.
[16] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. *Logic in Computer Science*, 1991.
[17] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. This volume, 1997.
[18] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. A Users Guide to HYTECH. Technical report, Department of Computer Science, Cornell University, 1995.
[19] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.
[20] G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.
[21] K. J. Kristoffersen, F. Larroussinie, K. G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *Proc. of the 7th International Joint Conference on the Theory and Practice of Software Development*, Apr. 1997.
[22] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 76–87, Dec. 1995.
[23] K. G. Larsen, P. Pettersson, and W. Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 575–586. Springer–Verlag, Oct. 1995.
[24] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. To appear in *International Journal on Software Tools for Technology Transfer*, 1997.
[25] F. Pagani. Partial orders and verification of real-time systems. In B. Jonsson and J. Parrow, editors, *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1135 in Lecture Notes in Computer Science, pages 327–346. Springer–Verlag, 1996.
[26] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
[27] A. Puri and P. Varaiya. Verification of hybrid systems using abstractions. In *Hybrid Systems Workshop*, number 818 in Lecture Notes in Computer Science. Springer–Verlag, Oct. 1994.
[28] R. Sedgewick. *Algorithms*. Addison-Wesley, 2*nd* edition, 1988.
[29] N. Shankar. Verification of Real-Time Systems Using PVS. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer–Verlag, 1993.
[30] A. Valmari. A Stubborn Attack on State Explosion. *Theoretical Computer Science*, 3, 1990.
[31] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of Symp. on Logic in Computer Science*, pages 322–331, 1986.
[32] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real–time transition systems. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224, 1993.
[33] W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.

A Proofs of Lemma 1, Theorem 2 and 3

Lemma 1 *Let G_1 and G_2 be zero–cycle free graphs such that ${G_1}^C = {G_2}^C$. If there is an edge $(x, y) \in G_1$ such that $(x, y) \notin G_2$, then $(G_1 \backslash \{(x, y)\})^C = {G_1}^C = {G_2}^C$.*

Proof Let α denote the edge (x, y) and let m be the weight of α in G_1. We will show that there is an alternative path in G_1 *not* using α with weight no more than m. From this fact the Lemma obviously follows.

As ${G_1}^C = {G_2}^C$, the shortest path from x to y in G_2 has weight no more than m. As $\alpha \notin G_2$, this path must visit some vertex z different from x and y. Now let m_1 be the shortest path–weight from x to z and let m_2 be the shortest path–weight from z to y; note that G_1 and G_2 agrees on m_1 and m_2, as they have the same shortest–path closure. Then clearly, $m \geq m_1 + m_2$.

Now assume that the shortest path in G_1 from x to z uses $\alpha = (x, y)$. Then, as a sub–path, G_1 will be a path from y to z. Since G_1 also has a path from z to y, it follows that G_1 will have a cycle from y via z back to y. The weight of this cycle can be argued to be no more than $(m_1 - m) + m_2$. However, as $m \geq m_1 + m_2$ and there are no negative cycles, this cycle must have weight 0 contradicting the assumption that G_1 is zero–cycle free.

Similarly, a contradiction with the zero–cycle free assumption of G_1 is obtained, if the shortest path in G_1 from z to y uses α. thus we can conclude that there is an path from x to y not using α with length no greater than m. $\square$

Theorem 2 *Let G be negative–cycle free graph. Then the shortest–path reduction of G is given by the graph:*

$$G^R := \left({G_\equiv}^R\right)^+$$

Proof We only prove that G^R is a candidate for a shortest–path reduction of G in the sense that $(G^R)^C = G^C$. The proof that G^R has minimal number of edges is left for the full version.

As all edges (x, y) of G^R have weight of the form $E_{G^C}(x, y)$, it is clear that for any path in G^R there is a path in G with same weight.

Now consider an edge (x, y) of G. We will demonstrate that there is a path in G^R with no greater weight.

If $x = min\, E_i$ and $y = min\, E_j$ for two $\equiv$–classes E_i and E_j, it follows that $E_{G_\equiv}(E_i, E_j) \leq E_G(x, y)$. Also, due to the property of reduction construction, there is a path in ${G_\equiv}^R$ between E_i and E_j with weight no greater than $E_{G_\equiv}(E_i, E_j)$. The same path, but now between min's of $\equiv$–classes, can be found in $({G_\equiv}^R)^+$. Thus, there is a path in G^R with weight no greater than $E_G(x, y)$.

If $x, y \in E_i$ for some $\equiv$–class E_i, an easy argument gives that $E_{G^R}(x, y) = E_{G^C}(x, y) \leq E_G(x, y)$.

Finally, consider the case when $x \in E_i$ and $y \in E_j$ for two different $\equiv$–classes, and assume that $E_G(x, y) = m$. Now let $m_1 = E_{G^R}(x, min\, E_i)$, $m_2 = E_{G^R}(min\, E_i, min\, E_j)$, and $m_3 = E_{G^R}(min\, E_j, y)$. Note that by the reduction construction $m_2 \leq E_{G^C}(min\, E_i, min\, E_j)$. Then there is a path in G^R from x to y via $\min E_i$ and $min\, E_j$ with weight $m_1 + m_2 + m_3$. Now, if $m < m_1 + m_2 + m_3$, there is a path in G from $min\, E_i$ to $min\, E_j$ of weight $m - m_1 - m_3 < m_2$ contradicting that m_2 is the weight of the shortest path in G between $min\, E_i$ and $min\, E_j$. Thus the path x, $min\, E_i$, $min\, E_j$, y in G^R has weight no greater than the edge (x, y) in G. $\square$

Theorem 3 *Every dynamic loop of a network of timed automata contains at least one covering state.*

Proof Assume a dynamic loop $L_d = (\bar{l}_1, D_1) \rightsquigarrow \ldots \rightsquigarrow (\bar{l}_k, D_k)$ with no covering states. However according to Proposition 2, L_d contains at least one entry node. Further, assume (without loss of generality) that the symbolic state $(\bar{l}, D) \in L_d$ is an entry node and the components $\bar{l}[1], \ldots, \bar{l}[m]$ of $\bar{l}$ are all in an entry node, and all the other components of $\bar{l}$, i.e. $\bar{l}[m+1], \ldots, \bar{l}[n]$, are not.

Now, we claim that if L_d contains no covering states, the set of components $\bar{l}_i[1], \ldots, \bar{l}_i[m]$ will remain in an entry node in all symbolic states $(\bar{l}_i, D_i) \in L_d$. Otherwise, if the set of local entry nodes changes, either grows or reduces, it will introduce a covering state. The case of growing is obvious due to the definition for covering states. The argument for the case of reducing is the same as the control nodes of all the components will reach $\bar{l}_1$ again by the end of L_d, meaning that the set will sooner or later grow again.

In fact, the assumption that L_d contains no covering states, implies an even stronger property, that is, all symbolic transitions in L_d are derived by components in $\bar{l}_i[m+1], \ldots, \bar{l}_i[n]$. A transition is derived by a local transition of a component in $\bar{l}[1], \ldots, \bar{l}[m]$, means that the set of local entry nodes will either grow or reduce (discussed above) or the local transition leaves the current entry node and enters an another entry node. The later case implies that the new entry node is a covering state.

Now we construct L'_d by removing $\bar{l}_i[1], \ldots, \bar{l}_i[m]$ from all symbolic states $(\bar{l}_i, D_i) \in L_d$, that is, L'_d contains only the components that are not in an entry nodes. Obviously, all the symbolic transitions of L_d are also in L'_d; thus L'_d must be a loop by definition. However, L'_d contains no components that are in an entry node. This contradicts Proposition 2. $\square$