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

A Loop Acceleration Technique to Speed Up Verification of Automatically-Generated Plans

Robert P. Goldman and Michael J.S. Pelican and David J. Musliner

SIFT, LLC; Minneapolis, MN, USA

Received: 2 April 2012 / Revised version: 18 February 2013

Abstract. The CIRCA planning system automatically creates reactive plans and uses formal verification techniques to prove that those plans will preserve system safety. CIRCA's timed automata verification system is highly efficient, yet can display pathologically bad behavior when reasoning about *reaction loops*, a particular form of interacting cycles of states. In this paper we describe a loop acceleration technique that recognizes these state space structures during the verification process and bypasses the process of expanding an arbitrarily large cycle of states, effectively compressing loops of arbitrary size into a compact, finite set of states. The resulting performance improvement can be very dramatic: in domains where tight loops of short-duration transitions interact with long-duration transitions, our new loop acceleration methods can reduce verification time (and hence planning time) from hours to below a second.

1 Introduction

The ability to automatically prove reachability properties of programs or plans has many applications in computer science, including safety proofs for plans generated by classical planning systems. Although intractable (or in some cases undecidable) in the worst-case, algorithmic improvements and modern computing hardware have made these model-checking techniques practical for larger and larger problems. Unfortunately, when pathological cases arise, practitioners may encounter the intractability.

We describe a specific challenge, *state space fragmentation*, that can arise in the context of model-checking *timed automata*, and present a practical solution to many instances of this challenge. The model of timed automata [1], an extension of nondeterministic finite state machines, was developed to analyze real-time systems, such as discrete supervisory control of temporally-extended processes. Systems for model-checking timed automata include UPPAAL [2–4] and Kronos [5].

The state-space fragmentation problem arises when state transitions of greatly different temporal latencies create interacting cycles of states. This problem was first described by Hune [6] and Iversen [7] in the context of verifying real-time controllers for LEGO® Mindstorms^{TM} robots. In some cases where two transition cycles apply to the same plan state, the shorter duration transition can "fragment" [8] the larger transition, creating an explosion in the number of states considered by the model-checker.

We encountered the fragmentation problem in the form of "reaction loops" in the CIRCA hard real-time planning, controller synthesis, and reactive execution system. CIRCA automatically synthesizes hard real-time controllers that may be modeled as timed automata, and uses a custom timed automaton model-checker in its controller synthesis [9–12]. As part of the controller synthesis process, the timed automaton model-checker is used to verify that the controllers are safe (that failure states are unreachable). We encountered the fragmentation problem checking CIRCA controllers that executed quick reactions to meet environmental threats while simultaneously monitoring long-duration processes. Such problems would typically make the verification runs take so long as to make the controllers impossible to analyze, and thus synthesis became impossible. We developed a new technique for *loop acceleration* that covers a case not handled by previous methods. We describe that technique and its implementation in this paper, and show that it radically speeds up timed automaton verification. While the technique exploits some particular features of CIRCA control problems, these features are not especially exotic, so the technique should be readily applicable in other timed automaton problems.

We begin by discussing related work on fragmentation in timed automata, focusing on the work by Hendriks and Larsen. After that, in Section 3, we introduce the CIRCA system, its controller synthesis process, and the way it uses a timed automaton verifier. We then review the model of timed automata, and discuss some of the most important features of timed automata reachability verification (Sections 4 and 5). We bring things together, discussing how the CIRCA controllers may be modeled with timed automata (Section 6). We introduce the problem of reaction loops (Section 7), and the loop acceleration technique that we have developed to handle the problem (Section 8). After that, we present a proof that analytically demonstrates the correctness of the technique (Section 9), and empirically demonstrate its usefulness with empirical results from tests with the CIRCA system (Section 10). We conclude with some ideas about future directions (Section 11).

2 Related work

The most closely-related work to ours is the work on loop acceleration for UPPAAL done by Hendriks and Larsen [13,8]. It was the discovery of this work that inspired our own. Despite that, their work is quite different from ours in technique and objectives.

In particular, their technique is particularly tailored to verifying *correct* controllers. It was developed in response to a problem verifying the correctness of a controller that conducted a busy-waiting loop during the course of a long-running process in the controller's environment. It was very expensive to recognize that the system would eventually reach the desired state — verification of an "eventually reachable" TCTL goal. Their solution involves *adding* loop acceleration edges to the model, allowing UPPAAL to detect the reachability more quickly, using breadth-first search, than with the original un-augmented model.

That approach would not help us in the verification of CIRCA control plans; in fact, it would actually make verification *worse*! The difference is that the primary task in CIRCA verification is to demonstrate that the controller is safe by showing that it can *never* reach an undesirable state. This means that verification involves exhaustive search of the state space, so that augmenting the model with new transitions (shortcuts), as Hendriks and Larsen do, would actually make the search space bigger, and in the case where the controller is safe, would make the search consume more time. Our technique, by contrast, works by collapsing together parts of the state space that are equivalent with respect to the class of safety queries, making exhaustive search faster.

Hendriks and Larsen's technique also works only in the case where a loop is concerned only with a single clock. That is, where all the guards and invariants involve only a single clock, and where the value of that single clock increases monotonically. By contrast, in our loops there are typically multiple clocks racing against each other, and clocks are reset — since the loop involves the controller repeatedly servicing some process in the environment.

Other researchers have recently presented approaches to acceleration in different, but related contexts. For example, Fietzke et al. [14] have shown how to accelerate loops involving a single clock in a variation of timed automata they call "Extended Timed Automata." Bozga et al. [15] present theoretical work on identifying periodic relations in model checking and an implementation in their FLATA toolset. However, they are interested in infinitely periodic loops, and are working with integer programs, rather than timed automata. Bardin et al. have developed a general theory of acceleration in model-checking [16], that includes loop acceleration as a special case of what they call "flat acceleration." They have applied it to integer programs in the FAST tool [17]. Closer to our own interests is the work by Ben Salah on partial order optimization for timed automata [18]. While the field of their optimization is different, it is similar in computing a convex clock region for multiple different paths, in their case paths that represent different permutations of the same discrete transitions. The theoretical underpinnings of this work might make available a more direct proof of correctness of our own loop acceleration.

3 CIRCA Background

CIRCA, the Cooperative Intelligent Real-time Control Architecture [9–12], like many autonomy architectures, comprises three layers:

1. The Adaptive Mission Planner (AMP)
2. The Controller Synthesis Module (CSM), and
3. The Real-Time Subsystem(RTS).

The AMP generates a long duration mission plan consisting of one or more mission phases. It constructs phases by reasoning about long term resource management (where computation time devoted to planning is a significant resource) and choosing groups of mission goals and threats to compose as planning problems for the CSM. Through on-line resource management and dynamic composition of planning problems, the AMP helps deal with the possibly intractable challenge of planning (and verifying) a single policy for the entire mission. This simplifies the overall autonomous control problem into a sequence of simpler problems for which controllers can more easily be planned, verified, and executed. The RTS executes the individual controllers, providing hard real time guarantees. We will not discuss these components further in this paper: the CSM is our focus here.

The CIRCA CSM contains two main modules of interest for this paper: the planner proper, which reasons

```
ACTION turn_on_main_engine              ;; Turning on the main engine
   PRECONDITIONS: '((engine off))
   POSTCONDITIONS: '((engine on))
   DELAY: <= 1

EVENT IRU1_fails           ;; Sometimes the IRUs break without warning.
   PRECONDITIONS: '((IRU1 on))
   POSTCONDITIONS: '((IRU1 broken))

;; If the engine is burning while the active IRU breaks,
;; we have a limited amount of time to fix the problem before
;; the spacecraft will go too far out of control.
TEMPORAL fail_if_burn_with_broken_IRU1
   PRECONDITIONS: '((engine on)(active_IRU IRU1) (IRU1 broken))
   POSTCONDITIONS: '((failure T))
   DELAY: >= 5
```

Fig. 1. Example transition descriptions given to CIRCA's CSM.

about time-abstract states to plan actions, and a verifier that reasons about partial and complete plans to ensure that they meet logical and timing safety requirements. In this section, we briefly sketch these functional modules and describe an example problem that we will carry throughout the paper, to illustrate how CIRCA uses model verification in the context of automated planning and how the loop acceleration technique speeds up model verification.

3.1 State Space Planning

Unlike traditional AI planners, CIRCA reasons about uncontrollable processes including adversaries, and metric, continuous time. The CSM takes in a description of the processes in the system's environment, represented as a set of time-constrained transitions that modify the value of world features. Discrete states of the system are modeled as sets of feature-value assignments. Transitions have preconditions, describing when they are applicable, and bounded delays, capturing the temporal characteristics of controllable processes (i.e., actions) and uncontrollable processes (i.e., world dynamics).

Example 1. Figure 1 shows several transitions from a CIRCA problem description for controlling the Cassini spacecraft during Saturn Orbit Insertion [19,20]. In this problem, CIRCA must generate a controller that will turn on the main engine in time to perform the orbital insertion within the time window. In order for the maneuver to complete successfully, the spacecraft must be guided by a working Inertial Reference Unit (IRU).

The transition descriptions, together with specifications of initial states, implicitly define the set of possible system states. The CSM is responsible for deciding, in each state, what action the system should take to maintain system safety and drive the system towards its goals. For example, Figure 2 illustrates a small portion of the Saturn problem's state space, after the CSM has made only its first few decisions about how to control the system.

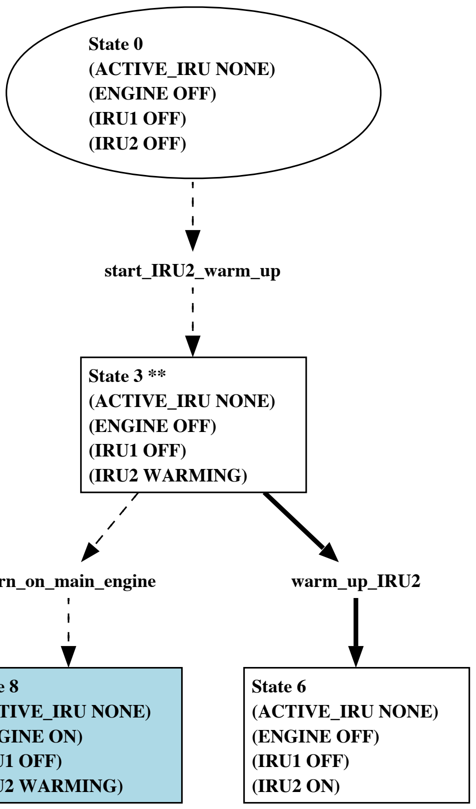


Fig. 2. The beginning of a state space plan for Saturn Orbit Insertion. In this diagram, the ellipse marks the initial state, other states are rectangles, and the shaded state is a goal state.

The CSM reasons about both controllable and uncontrollable transitions:

Action transitions represent actions selected by CIRCA; the CSM's objective is to assign an action to each reachable state. In Figure 2, a dashed arrow shows that the system has chosen the action `start_IRU2_-warm_up` in the initial state zero. Associated with each action is a worst case execution time, an *upper bound* on the delay before the action occurs.
The special "do nothing" action "no_op" can also be used. Typically this is done when the controller wishes to wait for some uncontrolled process to complete. For example, the controller might begin warming up an IRU, execute no-op in the state where the process of warming up is ongoing, and then designate the IRU as the active IRU, when it has completed warming up.

Temporal (uncontrollable) transitions represent uncontrollable processes. Associated with each temporal transition is a *lower bound* on its delay. If the preconditions hold true for at least this time, the transition may fire and enforce its postconditions. If a temporal transition leads to an undesirable state, the CSM may plan an action to *preempt* the temporal by ensuring that the action will definitely occur before the temporal could possibly occur. Transitions whose lower bound is zero are referred to as *events*, and are handled specially for efficiency reasons. Transitions whose postconditions include the distinguished proposition `(failure T)` are called *temporal transitions to failure* (TTFs).

Reliable temporal transitions represent continuous processes that may need to be employed by the CIRCA agent. Reliable temporal transitions have both upper and lower bounds on their delays. For example, when CIRCA turns on an Inertial Reference Unit it initiates the process of warming up that equipment; the process will definitely complete if it is continued without interruption for some time, as shown by the solid arrow leaving state 3 in Figure 2.

Note that each transition is an implicit description of many transitions in an automaton model. Each of these transitions is enabled in any discrete state that satisfies its preconditions, and disabled everywhere else. We explain how these implicit descriptions work more precisely in Section 6.

Central to the controller synthesis problem is *preemption.* If a temporal transition leads to an undesirable state, the CSM may plan an action to *preempt* the temporal:

Definition 1 (Preemption). A temporal transition may be preempted in a (discrete) state by planning an action for that state which will *necessarily* occur before the temporal transition's delay can elapse.

Note that successful preemption does not ensure that the threat posed by a temporal transition is handled; it may simply be postponed to a later state (in general, it may require a sequence of actions to handle a threat). A threat is handled by preempting the temporal transition with an action that carries the system to a state which does *not* satisfy the preconditions of the temporal.

3.2 CIRCA controller synthesis algorithm

Algorithm 1 (Controller Synthesis).

1. Choose a state from the set of unplanned reachable states (at the start of state space planning, only the initial states are reachable).
2. For each uncontrollable transition enabled in this state, choose whether or not to preempt it. Transitions that lead to failure states *must* be preempted. The CSM creates a boolean constraint variable for the preemption decision for each of these uncontrollable transitions.
3. Choose a single control action or `no-op` for this state.
4. Invoke the verifier to confirm that the (partial) controller is safe (see below for a discussion of how the verifier is invoked on partial controllers). "Safe," here is defined as "does not make any transitions to the distinguished failure state" and "successfully enforces all of the preemption decisions made in step 2."
5. If the controller is *not* safe, use a counterexample from the verifier to direct backjumping and return to step 1 [21].
6. If the controller *is* safe, recompute the set of reachable states.
7. If there are no unplanned reachable states (reachable states for which a control action has not yet been chosen), terminate successfully.
8. If some unplanned reachable states remain, loop to step 1.

The search algorithm maintains the decisions that have been made, along with the potential alternatives, on a search stack. The algorithm makes decisions at two points: step 2 and step 3. Preemption decisions are boolean: the algorithm can choose to require preemption or not, for each uncontrollable transition leading out of a state. The set of alternative action choices for a state is dependent on the domain description and several pruning heuristics that eliminate applicable but inappropriate actions from consideration.

Note that Algorithm 1 is implemented as an "on the fly" algorithm: it expands the reachable sub-space of the state space during its search. Unless forced to, it does not expand the full (discrete) state space, which can be exponential in the size of the input problem description.

3.3 Use of verifier

The CSM uses the verifier to confirm both that failure is unreachable *and* that all the chosen preemptions will be enforced. The CSM uses the verifier module after each assignment of a control action (see step 4). This means that the verifier will be invoked before the controller is complete. At such points we use the verifier as a conservative heuristic by treating all unplanned states as if they are "safe havens." Unplanned states are treated as absorbing states of the system, and any verification traces that enter these states are regarded as successful.

Treating unplanned states in this way constitutes a conservative heuristic in the sense that any detected failure is definitely a true failure (soundness), but the verifier may fail to identify failures that will later become apparent (incompleteness). At the limit, when the control program is complete, the verifier will be both sound and complete.

When the verifier indicates that a controller is *unsafe*, it will return a path from the initial state to the distinguished failure state. The action choices that the planner made for the states along that path form a set of candidate decisions to backtrack and revise, as discussed in [21]. If no safe plan is possible, the overall CSM may return failure, at which time the next higher level of CIRCA, the AMP, would try to compose a different problem for the CSM, perhaps by omitting some goals. The CSM algorithm will never return an unsafe controller.

To verify safety in CIRCA plans, the CSM represents the CIRCA control plans as timed automata. In the next section we review timed automata, and following that we explain the relationship between CIRCA control plans and timed automata.

We continue to work on improving CIRCA and applying it in new domains. Recent work has included research in using statistical verification to verify large CIRCA plans [22] and application to on-board autonomy for satellites [12,23].

4 Timed Automata

A timed automaton is a nondeterministic finite automaton (NFA) augmented with timing information [1] (our notation has been adapted from the descriptions in [24] and [25]).

Definition 2 (Timed Automaton). A timed automaton A is a tuple $\langle \mathcal{S}, s^i, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I} \rangle$ where

1. $\mathcal{S}$ is a finite set of locations;
2. s^i is the initial location;
3. $\mathcal{X}$ is a finite set of clocks;
4. $\mathcal{L}$ is a finite set of labels;
5. $\mathcal{E}$ is a finite set of edges; and
6. $\mathcal{I}$ is the set of invariants.

Each edge $e \in \mathcal{E}$ is a tuple $\langle s, L, \psi, P, s' \rangle$ where $s \in \mathcal{S}$ is the source, $s' \in \mathcal{S}$ is the target, $L \subseteq \mathcal{L}$ are the labels,[1] $\psi \in \Psi_{\mathcal{X}}$ is the *guard*, and $P \subseteq \mathcal{X}$ is a set of clocks to reset. Timing constraints ($\Psi_{\mathcal{X}}$) appear in guards and invariants. Guards dictate when the model *may* follow an edge, invariants indicate when the model *must* leave a state. In CIRCA models, all clock constraints on guards are of the form $c_i > k$ and all clock constraints on invariants are of the form $c_i \leq k$. In our models, all clock resets re-assign the corresponding clock to zero; they are used to start and reset processes.

[1] Labels are used in parallel composition of timed automata, which we do not use here.

The state of a timed automaton is a pair: $\langle s, C \rangle$. $s \in \mathcal{S}$ is a location and $C : \mathcal{X} \to \mathbf{Q} \geq 0$ is a clock valuation, that assigns a non-negative rational number to each clock.

A timed automaton *trace* is a sequence of state transitions that represents the computation of a timed automaton. Corresponding to any timed automaton, A, is a transition system, S_A, with two types of transitions: time-elapse transitions and discrete transitions:

Definition 3 (Time-Elapse Transition). A time-elapse transition, $\langle s, C \rangle \xrightarrow{t} \langle s, C + t \rangle$ can occur when for all t' such that $0 \leq t' \leq t$, $C + t'$ satisfies the invariant $I(s)$.

Definition 4 (Discrete Transition). A discrete transition, $\langle s, C \rangle \xrightarrow{e} \langle s', C' \rangle$, for some $e \in \mathcal{E}$ can occur when C satisfies the guard of e, $\psi(e)$, and C' satisfies the reset of e applied to C, $P(e, C)$.

For brevity, we will refer to discrete transitions as *jump transitions* or *jumps* in the remainder of this paper.

To understand CIRCA's CSM graphs, the concept of time abstraction is helpful.

Definition 5 (Time abstraction). The time abstraction of a timed automaton is a non-deterministic finite automaton whose states correspond to the locations of the timed automaton, and in which there is an edge e between s and s' whenever there exists s'' and t such that $s \xrightarrow{t} s'' \xrightarrow{e} s'$.

The CIRCA CSM graph is the time abstraction of a TA model of the corresponding plan. The translation of CSM graphs to TA models is described in Section 6.

Figure 3 illustrates the timed automaton model corresponding to our running example, the partial Saturn orbit insertion plan shown in Figure 2. Since the CSM has not yet completed the plan in Figure 2, the timed automata model has sinks at locations 4 and 5, corresponding to the unplanned CSM states 6 and 8. Figure 4 illustrates the corresponding transition system reachability graph, where boxes correspond to a reachable location and clock zone, represented as a difference bound matrix. The reachability graph shows that locations 4 and 5 are reachable, but since their corresponding states are unplanned, the verification traces halt there. The plan is safe so far, since failure is not reachable.

In this paper, we will concern ourselves primarily with *reachability verification* of a particular timed automaton. We will be asking if it is possible for a timed

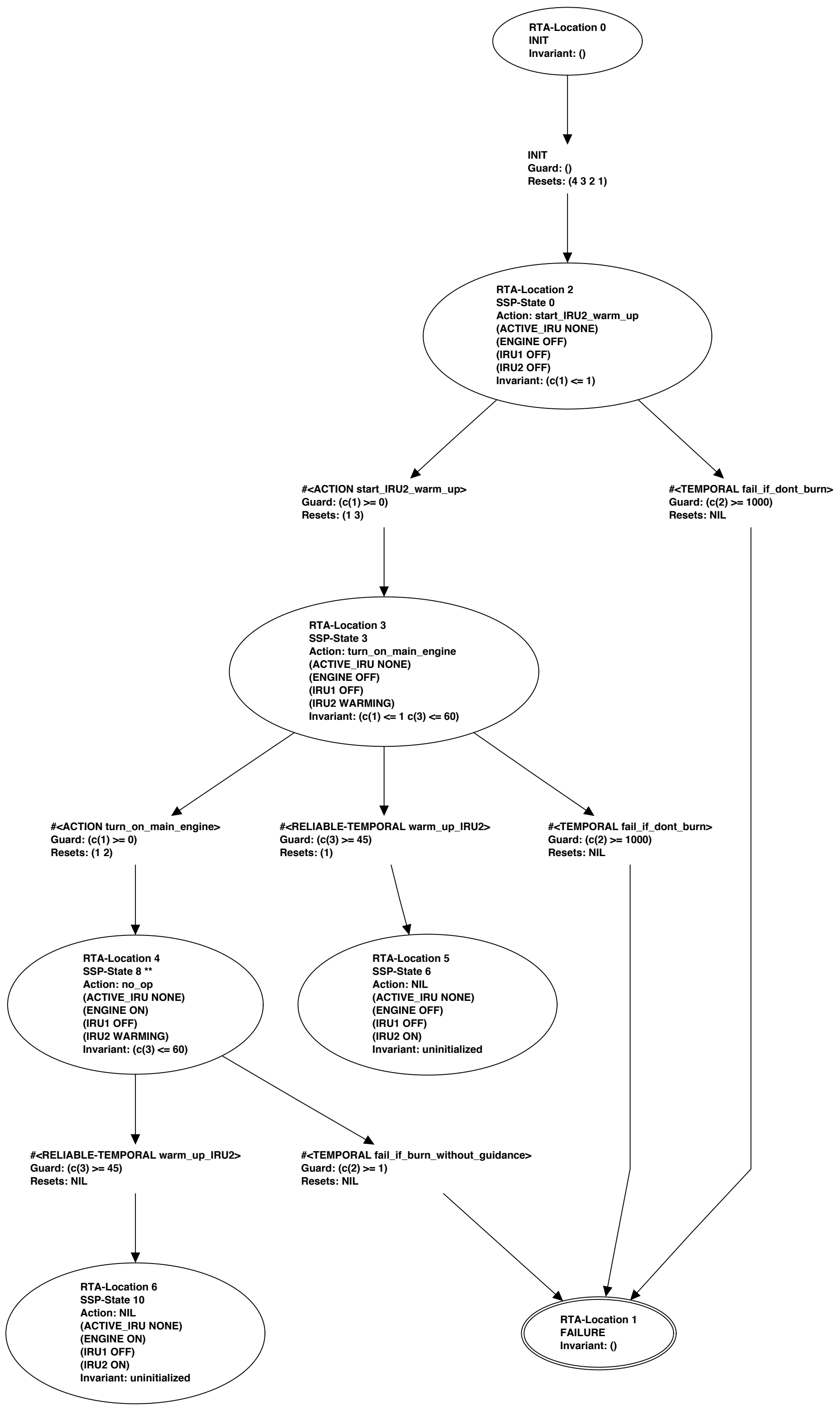
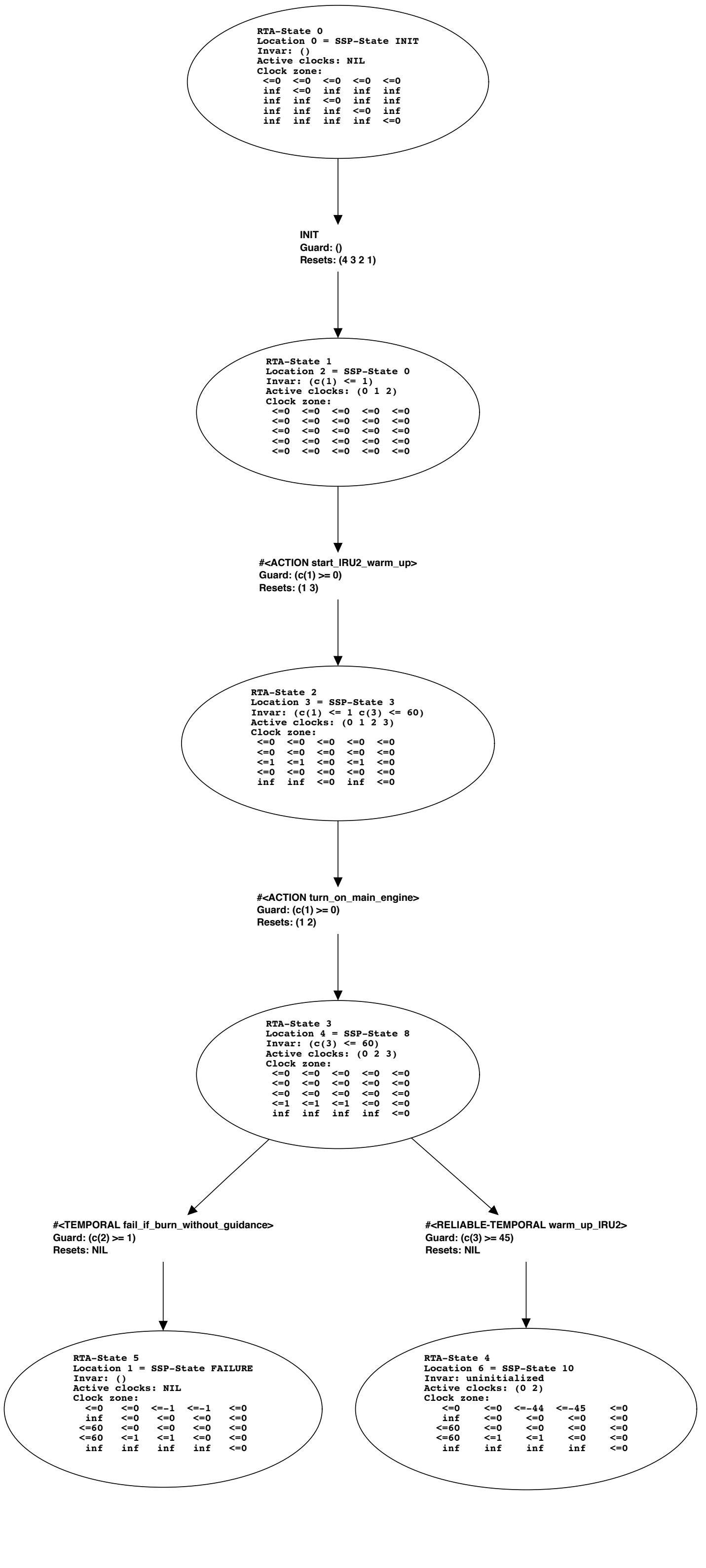


Fig. 3. The timed automaton model corresponding to Figure 2.



```
RTA-State 0
Location 0 = SSP-State INIT
Invar: ()
Active clocks: NIL
Clock zone:
 <=0  <=0  <=0  <=0  <=0
 inf  <=0  inf  inf  inf
 inf  inf  <=0  inf  inf
 inf  inf  inf  <=0  inf
 inf  inf  inf  inf  <=0

INIT
Guard: ()
Resets: (4 3 2 1)

RTA-State 1
Location 2 = SSP-State 0
Invar: (c(1) <= 1)
Active clocks: (0 1 2)
Clock zone:
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0

#<ACTION start_IRU2_warm_up>
Guard: (c(1) >= 0)
Resets: (1 3)

RTA-State 2
Location 3 = SSP-State 3
Invar: (c(1) <= 1 c(3) <= 60)
Active clocks: (0 1 2 3)
Clock zone:
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0
 <=1  <=1  <=0  <=1  <=0
 <=0  <=0  <=0  <=0  <=0
 inf  inf  <=0  inf  <=0

#<ACTION turn_on_main_engine>
Guard: (c(1) >= 0)
Resets: (1 2)

RTA-State 3
Location 4 = SSP-State 8
Invar: (c(3) <= 60)
Active clocks: (0 2 3)
Clock zone:
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0
 <=0  <=0  <=0  <=0  <=0
 <=1  <=1  <=1  <=0  <=0
 inf  inf  inf  inf  <=0

#<TEMPORAL fail_if_burn_without_guidance>
Guard: (c(2) >= 1)
Resets: NIL

#<RELIABLE-TEMPORAL warm_up_IRU2>
Guard: (c(3) >= 45)
Resets: NIL

RTA-State 5
Location 1 = SSP-State FAILURE
Invar: ()
Active clocks: NIL
Clock zone:
   <=0   <=0  <=-1  <=-1   <=0
   inf   <=0   <=0   <=0   <=0
  <=60   <=0   <=0   <=0   <=0
  <=60   <=1   <=1   <=0   <=0
   inf   inf   inf   inf   <=0

RTA-State 4
Location 6 = SSP-State 10
Invar: uninitialized
Active clocks: (0 2)
Clock zone:
   <=0    <=0  <=-44  <=-45    <=0
   inf    <=0    <=0    <=0    <=0
  <=60    <=0    <=0    <=0    <=0
  <=60    <=1    <=1    <=0    <=0
   inf    inf    inf    inf    <=0
```

Fig. 4. The timed automaton reachability space corresponding to Figure 2.

automaton to reach a particular location, $s \in S$. In particular, we will be checking the safety of a CIRCA plan by asking if a timed automaton corresponding to the plan can ever reach the distinguished failure state. While such reachability queries are not tractable, they are computable, and can be answered by simple graph search algorithms.

Algorithm 2 (Reachability Verification).

```
 1: let $openlist \leftarrow \{\langle s^i, \mathbf{0}\rangle\}$ {initial state}
 2: **while** $openlist \neq \emptyset$ **do**
 3:    $s \leftarrow$ pop($openlist$)
 4:    **if** visited($s$) **then**
 5:       skip
 6:    **else if not** action-assigned($s$) **then**
 7:       skip {For incremental verification, see below}
 8:    **else if** failure($s$) **then**
 9:       **return unsafe**
10:    **else**
11:       let $succ \leftarrow$ successors($s$)
12:       $openlist \leftarrow openlist \cup succ$
13:    **end if**
14: **end while**
15: **return safe**
```

Of course, any naive attempt to apply Algorithm 2 is doomed to failure. In particular, if one assumes dense time, the state space of this search may be uncountably large. Practical verification systems for timed automata typically search in a space of equivalence classes of states, since the state space of any timed automaton can be reduced to a finite number of equivalence classes [26]. Typically, a verification system will collapse together multiple states using clock *zones*. In the following discussion we will use "state" for both state and state equivalence class; no confusion should result since any practical algorithm will have to manipulate the latter, rather than the former.

Verification systems also employ clever techniques for reducing the number of states that must be explored, answering the visited query (step 4, above), and computing the successor set (step 11). Furthermore, instead of simply returning **unsafe**, reachability verification systems typically return a *counterexample trace*, that exhibits a path from the initial state to the failure state, and can be used for debugging. To the best of our knowledge, CIRCA is unique in automating the exploitation of counterexample traces in planning (see [21] for an explanation of our technique for using counterexample traces to direct backtracking in planning search). We will return to the skeletal search algorithm later and describe modifications for our planning application.

Recall that the CSM verifies partial plans during construction, before they are fully designed. Before the planning process is complete, there will be states that do not yet have action assignments. We verify partial plans by treating such states as safe sink states. That is the function of the check in line 6 of Algorithm 2. Note that when the planning algorithm (Algorithm 1) is completed, all of the states will have an action assigned to them, so the final verification will be a full verification. The sequence of verifications can be thought of as a fixpoint computation that converges on a full TA verification of the CSM plan.

5 Difference-Bound Matrices

As mentioned in Section 4, CIRCA's verifier reduces the size of its search space by collapsing states to equivalence classes with respect to their temporal characteristics. For example, given two automata states, $\langle s, C_1\rangle$ and $\langle s, C_2\rangle$, in the timed automata representing a CIRCA plan graph, if the clock assignments C_1 and C_2 differ only in a single clock value, and that value exceeds any edge's test (guard or invariant) on the clock, then $\langle s, C_1\rangle$ and $\langle s, C_2\rangle$ can be treated as members of a single equivalence class. Similarly, if C_1 and C_2 differ only in the value of a single clock x_i, and x_i is only tested against 1 and 3, it may be possible to collapse C_1 and C_2 into equivalence classes based on the inequalities $x_i < 1$, $1 \leq x_i < 2$, and $2 \leq x_i$. Since clock values are constrained to increase uniformly, it is also straightforward to collapse states in which multiple clocks differ, as long as relevant constraints are satisfied.

To support the required equality operation, the CIRCA verifier uses the difference-bound matrix (DBM) [27, 25] data structure to represent the set of clock values in an automata state. To understand the nature of the "fragmentation" problem and our "loop acceleration" solution it is helpful to understand the basics of the DBM structure.

Each DBM is a $(k+1)\times(k+1)$ matrix D representing a clock zone for a state in the timed automaton for a CIRCA graph. D will have k clocks $x_1, ...x_k$, representing $k-1$ uncontrolled transitions and the special controlled action clock x_1. For each i, D_{i0} is the upper bound on the absolute value of clock x_i and D_{0i} is the lower bound on the value of the clock x_i. The other entries in the matrix represent bounds on the relative distance between two clock values. For all i and j for which we have clocks x_i and x_j, the DBM entry D_{ij} is the upper bound on the difference between values of clocks x_i and x_j. To allow for strict and non-strict bounds, each entry also contains a second symbol (0 or 1) to indicate whether the inequality of the bound between the clocks is strict or not. The absence of a bound between x_i and x_j is represented by ∞ in D_{ij}. Thus, the domain, $\mathbb{D}$, of the D_{ij} entries is $\mathbb{Q} \times \{0,1\} \cup \infty$.

Thus, a DBM D is a $(k+1)\times(k+1)$ matrix whose entries are elements from $\mathbb{D}$. A clock interpretation v satisfies D iff for all $1 \leq i \leq k, x_i \leq D_{i0}$ and $-x_i \leq D_{0i}$, and for all $1 \leq i,j \leq k, x_i - x_j \leq D_{ij}$. Although there are more than one DBM describing any clock zone, there is an algorithm for computing an unique, canonical DBM for any clock zone. In order to implement Algorithm 2, we must compute a *successor* zone for any DBM D progressing over an edge e, which can by achieved by a combination of *intersection*, *time-elapse*, and *projection* operations on clock zones.

Note Readers with an AI or constraint programming background may notice the similarity between DBMs and Simple Temporal Networks (STNs) [28]. These two data structures represent the same class of constraints, and both support emptiness (consistency) checks, however, they are implemented differently, in order to optimize different operations. STNs are typically used in planning and scheduling problems where a set of temporal variables are related together by an increasing set of constraints. These constraints accumulate as the planning or scheduling problem is solved. Often there is backtracking, which involves the removal of constraints, and often the set of variables is not fixed: new time points may be added to the network. With DBMs used in timed automaton verification, on the other hand, the set of variables is typically fixed, and there is no backtracking. Operations to be optimized include checking to see if one DBM is identical to another or contains another. These checks are typically not needed in planning and scheduling applications.

6 Modeling CIRCA with Timed Automata

Recall that as part of the CIRCA controller synthesis process, we *incrementally* verify the quality of partial plans, as they are generated (see step 4 of Algorithm 1). To do so, we translate the CIRCA problem representation into timed automata, which involves expanding the implicit representation of the state space.

6.1 Translating CIRCA problems to timed automata

Recall that CIRCA controller synthesis problems are posed by providing an initial state description, and a set of transition descriptions, partitioned into volitional (controlled) and non-volitional (uncontrolled) transitions. Taken together, these specify a timed automaton, as follows.

Definition 6 (CIRCA problem). A CIRCA planning problem is a 4-tuple, $\mathcal{P} = \langle \mathcal{F}, \mathcal{V}, \mathcal{T}, i\rangle$ In it, $\mathcal{F} = \{f_0...f_m\}$ is a set of state features. For each $f_i \in \mathcal{F}$, $\mathcal{V}_{f_i} = \{v_{i0}...v_{ik_i}\}$ is a set of possible values. The initial state specification, i, is a *full feature assignment* (see Definition 7 below).

$\mathcal{T} = \mathcal{A} \cup \mathcal{U}$ is a set of CIRCA transitions (defined further below), that is partitioned into controllable ($\mathcal{A}$) and uncontrollable (u) transitions. For historical reasons, these are sometimes referred to as volitional and non-volitional transitions.

All CIRCA problems contain a distinguished failure feature, whose possible values are $\top, \bot$ (true and false). Failure is always false in the initial state. Note that any simple safety property, $\Box\phi$, or $\Box\neg\phi$ can be captured with use of the failure feature. More complex safety properties can be encoded by compiling them into the transition definitions, but we rarely have the need to do this.

Definition 7 (Feature Assignment). A feature value assignment, $\langle f, a\rangle$, maps a feature to one of its possible values. A feature assignment, $\mathcal{A}$, is a set of feature value assignments for a subset of $\mathcal{F}$. The set of features whose values are specified by $\mathcal{A}$ is $\text{feats}(\mathcal{A})$. A full feature assignment is a feature assignment where $\text{feats}(\mathcal{A}) = \mathcal{F}$. Finally, we can project a feature assignment onto a subset of features, $\mathcal{G} \subseteq \mathcal{F}$: $\pi_{\mathcal{G}}(\mathcal{A})$, by removing the elements of $\mathcal{A}$ whose features are not in $\mathcal{G}$.

Definition 8 (CIRCA transition description). A CIRCA transition is a 4-tuple, $t = \langle N, P, E, \Delta\rangle$. N is the name of the transition. P, the *precondition*, is a feature assignment, as is E, the *effect* of t.[2] Δ is a set of time bounds on the occurrence of the transition, a pair, $\langle \text{lb}(t), \text{ub}(t)\rangle$, of lower and upper bound, which we will discuss further below. For technical reasons, we require that P and E be *inconsistent*: E must assign a different value to at least one of the features in P.

The set of locations of the timed automaton corresponding to $\mathcal{P}$ is defined by $\mathcal{F}$ and $\mathcal{V}$: it is the set of all possible full feature assignments. Each location, l corresponds to a unique feature assignment, which we will write as $\text{fa}(l)$.

To complete the construction, we need additional definitions to translate the CIRCA transition descriptions into sets of transitions in the timed automaton. The first is the definition of applicability of a CIRCA transition:

Definition 9 (Applicability of CIRCA transition). A CIRCA transition, t, is applicable at a location, l, when the feature assignment of l satisfies the preconditions of t: $\text{fa}(l) \models P(t)$.[3]

The set of transitions that are applicable at l is $\text{app}(l)$. We will have need of the set of uncontrollable transitions at l, $\text{app}_{\mathcal{U}}(l) = \text{app}(l) \cap \mathcal{U}$.

Then we must define the translation of each CIRCA transition to one or more edges in the timed automaton:

Definition 10 (Image of CIRCA transition). The image of a CIRCA transition, t, from location l, $\text{img}(t, l) \rightarrow l'$, is the location l' such that:

$$\text{fa}(l') = \big(\text{fa}(l) - \pi_{\text{feats}(E(t))}(\text{fa}(l))\big) \cup E(t)$$

[2] CIRCA actually supports nondeterministic transitions that have multiple different effect feature assignments, but for this presentation we ignore that complication.

[3] Because of the simplicity of the feature-value representation, for any two assignments, $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1 \models \mathcal{A}_2$ iff $\mathcal{A}_1 \supseteq \mathcal{A}_2$.

I.e., the successor state is updated by the effects of t.

Definition 11 (Clocks for CIRCA transitions). For each *uncontrollable* CIRCA transition, t, we define a unique clock, $c(t)$. Note that a single CIRCA transition will correspond to multiple edges in the timed automaton.

There is a single clock, c_{RTS}, that governs the function of all of the controlled actions.

Finally, we need the definition of a CIRCA *plan*, or controller design:

Definition 12 (CIRCA plan). A CIRCA plan for $\mathcal{P}$, $\Pi_{\mathcal{P}}$, is a partial function from locations of $\mathcal{P}$ to controllable transitions or no-op. For brevity, we will simply use Π in the following.

The CIRCA plan is similar to other AI planning systems, and similar to deterministic Markov Decision Process policies, in assigning a single action to each state. In this way it diverges from the discrete controller model of Ramadge and Wonham [29] which *disables* some subset of the controllable transitions for a state.

Definition 13 (Translation of CIRCA transition). The translation of a CIRCA transition, t at a location, l is:

$$\phi(t, l) = \langle t, l, \text{clock}(t) \geq \text{lb}(t), \mathcal{R}(l, \text{img}(t, l)), \text{img}(t, l)\rangle$$

if t is applicable in l, otherwise nothing.

The set of clocks reset by the timed automaton edge, $\mathcal{R}(l, \text{img}(t, l))$, is as follows:

1. For each uncontrolled transition, t' that is applicable at $\text{img}(t, l)$ and that is not applicable at l, add $\text{clock}(t')$;
2. If $\Pi(\text{img}(l)) \neq \Pi(l)$, add c_{RTS}.

All resets are to zero.

To complete the construction, we need to specify the invariants of all of the locations in the timed automaton:

Definition 14 (Location invariants). The location invariant of l, $I(l)$, is the following conjunction:

$$\left(\bigwedge_{t \in \text{app}_U(l)} \text{clock}(t) \leq \text{ub}(t)\right) \wedge X$$

Where X is $c_{\text{RTS}} \leq \text{ub}(\Pi(l))$ unless $\Pi(l) = \text{no-op}$, in which case $X = \top$.

We notate the resulting translation as $\theta(\mathcal{P})$.

A CIRCA plan, Π, is *complete* if the set of locations in its domain is a subgraph of the timed automaton that is closed under reachability.

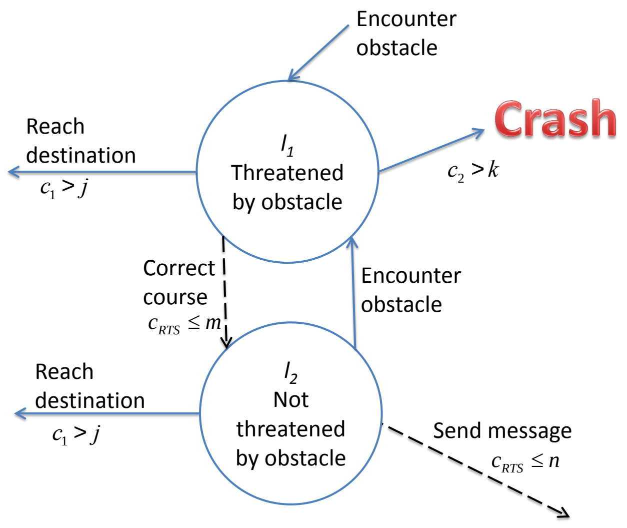


Fig. 5. An example of the pathological situation causing state-space fragmentation. Controlled actions are shown as dashed edges, uncontrolled as solid.

6.2 Verifying CIRCA plans

In order to verify a (partial) plan, Π, we translate it into a timed automaton (TA) model, $\theta(\Pi)$, using the above construction. We search the resulting model to verify the proposition $\neg \diamond$ failure. We do this using an "on the fly" construction method so that we only build a reachable subspace of the timed automaton.

As mentioned earlier, we conduct this verification multiple times, while constructing the plan, per our discussion of incremental verification in Section 3.3. To do so, we must verify partial plans, as well as complete plans. While doing so, any state on the frontier of $\theta(\Pi)$ – i.e., a state that is reachable from an element of Π but that has not yet been assigned an action or no-op, is made an absorbing state. This has the effect of treating these states as safe havens, giving us an over-approximation of the safety of the plans, which converges on a true verification.

7 The Problem: Reaction Loops

Some patterns of interaction between a CIRCA controller and its environment, in a CIRCA plan, cause state space explosion when verifying. Hendriks and Larsen [8] refer to this state space explosion as *fragmentation*, because it is a failure of the timed automaton verifier's abstraction methods in which the verifier cannot exploit repeated structure. Fragmentation occurs when a high speed process interacts with a slow one. In control systems, this typically occurs when a digital control system (fast) interacts with its environment (slow).

Examples arise in the following situations: CIRCA is controlling a system in an environment that presents the system with repeated threats, while a slow process carries the system towards a state where it can achieve its goals. For example, a vehicle might have to carry out small course corrections in response to obstacles in its path, while it is navigating to its destination. See the CIRCA-generated controller design in Figure 5. The system can rapidly respond to the need for a course correction, where "rapidly" means that the duration of the response transition is small relative to the duration of the process of reaching the destination, and the need for course corrections can also recur frequently. One more complication is necessary — there must be another transition out of the "unthreatened" state (the one where no course connection is necessary), that will impose an invariant on that state. For example, while the vehicle is navigating smoothly, but before it has reached its destination, it might wish to seize the opportunity to send a message to base that will update information about its current state.

Intuitively, what happens during verification is that the verifier first considers what happens if it must correct its course early in the course of a traversal, and then later, and then later, and then later, and . . . The verifier considers every way that the clocks representing the course correction processes might interact with the clocks for the navigation process and opportunity exploitation. Consider what happens during forward search, when the system considers the state where the vehicle is threatened with collision (l_1), and is navigating towards the destination. At this point, the clock on "reach destination," c_1, starts at zero. After j time units expire on this clock, the system may make a discrete transition corresponding to reaching its destination. If the vehicle does not take corrective action before time k, it may crash. More formally, when entering l_1, c_2 is reset to zero, and after k time units, the Crash jump is enabled.

In this threatened state, the controller takes an action to correct its course. This action has some duration, m, much smaller than j. That duration imposes an invariant on l_1; the system cannot remain in l_1 more than m time units before transitioning. Reaching the bottom location, l_2, the controller attempts to send a message, an action that takes n time units. The maximum delay, n, also imposes an invariant, this time on location l_2. n is also much smaller than j. This action may succeed, or the vehicle may encounter an obstacle before n time units elapse, returning the system to l_1. In the resulting state, of the system the location will again be l_1, and $c_1 \leq m + n$. The next time around the loop, $c_1 \leq 2(m + n)$, and so forth. When j is large relative to m and n, the resulting number of search states in the verifier can be very large, and can make reachability testing infeasible.

The clock zone techniques for collapsing the temporal state space [26] fail to partition the state space into a small number of equivalence classes, and the state space becomes *fragmented*. In practice, this problem can cause

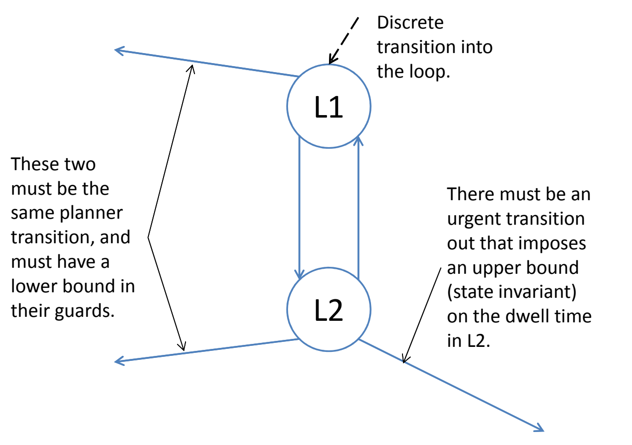

Discrete transition into the loop.
L1
L2
These two must be the same planner transition, and must have a lower bound in their guards.
There must be an urgent transition out that imposes an upper bound (state invariant) on the dwell time in L2.

Fig. 6. An illustration of the subgraph pattern (in the timed abstraction graph) that provides candidates for loop acceleration in CIRCA.

CIRCA to fail to find plans in important cases — the reaction loop problem is not at all uncommon in safety-focused controllers. Note that if there is no invariant in the second state of the loop, like the one provided by "send message," if all we are concerned with is reaching the destination *eventually*, then the pathology does not arise; clock zone techniques (based on difference-bound matrices) are sufficient.

Note that the number of loops is not relevant to the safety of the controller. For the purpose of checking safety, the only two things that matter are (1) whether the system can follow the "crash" transition and (2) what is the state of the clocks when it leaves the loop through either reaching the destination or sending a message. The first question can be answered by simple static inspection, whether the k lower bound on crashing is greater or less than the m upper bound on evasive maneuvers. We will not consider this question further in the paper. Our focus will be on efficiently answering the second question, noting that the number of iterations around the loop, and the individual dwell times in each location are of no interest in and of themselves.

8 Loop Acceleration for Reaction Loops

In this section, we discuss how we optimize away the state space fragmentation we just described. In CIRCA's TA verifier, while performing the search described in Algorithm 2, we look for instances of the pattern of states and transitions forming a reaction loop that is a candidate for acceleration. Then we rewrite the successors so that we generate just three states per loop, no matter how many iterations the loop makes, in a kind of local optimization.

In the course of verification, our technique looks for candidates that meet the reaction loop pattern described in the previous section. We show an example of this pattern in Figure 6. Specifically, in the forward verification search, the verifier checks for states where:

1. There is a backedge from the current location, to the location from which the current state was reached.
2. There is a long-duration process with a lower bound on its completion time (in CIRCA terms, a temporal) that is active in both of the locations.
3. There is a CIRCA transition in the current location, *other than the backedge*, that imposes an upper bound (invariant) on the state dwell time.

More formally, when checking a successor, s, for location l, reached by transition t, we check the following conditions:

1. Backedge: $\exists t' | t' = l \to l'$ and $t = l' \to l$;
2. Long duration process: There exists a CIRCA transition, T, such that
$$\exists t', t'' | t = l' \to l, t' = \phi(T, l), t'' = \phi(T, l')$$
and the lower bound on the clock for T is "large," where "large" is a threshold that may be set as a parameter of the optimization.
3. Upper bound on the state dwell time: $\mathcal{I}(l) \neq \infty$.

The presence of the backedge (1), of course, is necessary for there to be a loop that we can accelerate. However, while necessary, this is not sufficient. If we have only the backedge and the long duration process (2), then the difference-bound matrices will efficiently abstract the state space of the timed automaton, and there will be no fragmentation. It is only if we add the separate upper bound (3), that we encounter fragmentation.

In the context of the CIRCA solver, the very large loop-related state space ordinarily generated by a reaction loop can be collapsed into only three states:

1. A state for the top location, entering the loop;
2. A state for the bottom of the loop location and
3. A state for the top location, in iterations after the first entry.

The reason that this is possible is that we can compute the possible states of the various clocks upon leaving the loop without explicitly enumerating the states relating to the two clocks that control the loop proper. The loop clocks will behave the same way upon exit from the loop no matter how many times the loop is executed (with the exception of the first entry into the loop, which is why we have *three* states, instead of two. We may reason by cases about all of the other clocks. During the loop, the other active clocks will be related to each other (and to the loop clocks) in one of two ways:

1. They will be synchronized (possibly with some offset), when they enter the loop, and will stay synchronized or
2. They will be unconstrained with respect to each other.

Note that when we say "synchronized (possibly with some offset)" what we mean is that the clocks will be at some bounded distance from each other — not necessarily zero — and that this distance will not change, no matter how many times they go through the loop.

Which of the two possibilities — clocks stay synchronized, or clocks may stray an arbitrary distance — obtains depends on what happens when the system passes through the transition that corresponds to the backedge. All of the clocks that are reset by this transition will be reset to zero and will become (perfectly) synchronized. The others will be unaffected, so the relations among them (synchronized or not) will remain, and they will be desynchronized from the clocks that are reset. Note that this simplification occurs partly because of three special features of CIRCA timed automata:

1. Each discrete transition is controlled by exactly one clock, with a guard that is controlled by constants. There may additionally be state invariants, that come from urgent discrete transitions: these are also tested against constants. That is, there are no constraints that *directly* relate two clocks together (although some such constraints may be inferred in the closure computations over the DBMs).
2. When a discrete transition is followed, the clock controlling that transition will either be reset, or will become inactive.
3. The verification effort aims at checking reachability of states (in our case, the distinguished failure state) that are not included in any loop.

These are not requirements of the TA model, but we believe that they are common features of many control systems where a discrete controller must interact with temporally-extended processes.

The DBM implementation of timed automata substantially reduces the state space by reasoning efficiently with equivalence classes of clock valuations. The combination of the special features of CIRCA and the loop structure allow us to collapse the TA state space even further, and construct equivalence classes that are larger than the ones already captured by the DBM abstraction. We may simply remove the upper bounds on any clock that is not reset in the loop since it can grow without limit until the system exits the loop. We still enforce the state invariants imposed by transitions leaving the loop, so clock values cannot grow to values inconsistent with urgent transitions leaving the loop. We enforce the synchronization constraints which do not change between loop iterations, and thus capture the exit conditions (the state of the clocks upon leaving the loop) very simply in difference-bound matrices. We also do a simple static check to verify that the state of the system *in* the loop remains safe, as well.

Loop acceleration procedure The actual loop acceleration is done by a local, "peephole" optimization during forward state space verification search. In Algorithm 2, when we create the successors to state, in line 11, we check each successor to see if it is a candidate for loop acceleration, according to the criteria outlined above. If the successor is a candidate, we replace it according to Algorithm 3. In particular, our algorithm will rewrite two states for each loop: it will initially detect the opportunity to accelerate when the search reaches a state corresponding to the bottom of the loop. At that point it will recognize the backedge and accelerate the state corresponding to reaching the location at the bottom of the loop. Then it will create a second accelerated state corresponding to returns to the top of the loop.

Algorithm 3 (Loop acceleration rewrite).

```
1:  **proc** accelerate search state($s, t$) {State $s$ reached via $t$}
2:  let $r \leftarrow P(t)$ {clocks reset}
3:  let $nr \leftarrow$ activeclocks($s$) $- r$ {clocks *not* reset}
4:  **for all** $c \in nr$ **do**
5:      **for all** $c' \in r$ **do**
6:          desync(dbm($s$), $c, c'$)
7:      **end for**
8:      {Remove upper bound: $0 \leq c < \infty$}
9:      dbm($s$)[$0, c$] $\leftarrow (0, 1)$
10:     dbm($s$)[$c, 0$] $\leftarrow \infty$
11: **end for**
12: {now conduct normal DBM updates ...}
13: dbm($s$) $\leftarrow$ dbm($s$) $\cap \mathcal{I}(s)$)
14: dbm($s$) $\leftarrow$ elapse_time(dbm($s$))
15: dbm($s$) $\leftarrow$ dbm($s$) $\cap \mathcal{I}(s)$)
16: **if** empty(dbm($s$)) **then**
17:     **return false** {infeasible}
18: **else**
19:     **return true** {feasible}
20: **end if**
```

Recall that P is the set of clocks reset by a timed automaton transition. $\mathcal{I}(l)$ is the invariant for a timed automaton location, l; we extend it to search states by projection: $\mathcal{I}(s) = \mathcal{I}(l)$ for $s = \langle l, C \rangle$. Notation is as per Definition 2. dbm(s) is the difference bound matrix corresponding to the search state s. Clocks in the procedure are counting numbers; c_0 represents the numerical zero value. The operations in lines 13-15 are the standard operations for computing a successor difference bound matrix, per Alur's survey article [25].

The desync($\cdot, \cdot$) procedure desynchronizes its two argument clocks. It does this by relaxing constraints on the distances between the two clocks:

```
**proc** desync($dbm, c_1, c_2$)
$dbm[c_1, c_2] \leftarrow \infty$
$dbm[c_2, c_1] \leftarrow \infty$
```

The modifications of standard TA successor DBM computations in Algorithm 3 are above line 13: clocks for transitions that are *not* reset in the back-edge transitions are desynchronized from clocks that are, and additionally their upper bounds are removed. The intuitive justification for these modifications is the case analysis above: the desynchronization only affects the relationship between clocks that are reset by the back edge, and those that are not. Clocks that are not reset will retain the interrelationships that were established on entry into the loop. We justify this intuition in the next section.

Example 2. Consider how the optimization would be applied to the situation in Figure 5. In the verification search, we encounter l_1, which is processed normally:

The verifier will try to generate a successor for reaching the destination. The outcome of this is not of interest for our discussion; we pass over it. Next, the verifier will attempt to generate a successor for the "Crash" state. If it succeeds, verification will terminate with a counterexample, so let us assume that it does not (*i.e.*, that $m < k$). Finally, the verifier will generate a successor for l_2.

At this point, the verifier will detect the possibility of loop acceleration, and will perform the modifications described in Algorithm 3 to the DBM of the corresponding search state. It will generate successors for reaching the destination, and sending a message, assuming that they are consistent with the temporal constraints. It will also generate a successor for the back-edge, "encounter obstacle," that returns the system to l_1. When it does so, it will apply the same optimization to the DBM of the search state. It will attempt to generate out edges as per normal, but the successor state for l_2 will be recognized as already closed, so no further loop states will be generated.

The current version of CIRCA incorporates our loop acceleration technique as an option. On many scenarios this optimization speeds up problem solving sufficiently to make previously infeasible problems solvable. In the next two sections, we demonstrate the correctness of the algorithm, then present test results.

9 Proof of Correctness

In order to prove our technique correct, we must show that it does not change the outcomes of any verification. We do this by showing that for any trace, T, passing through an accelerated loop, that is found by the standard verification algorithm, there exists a corresponding trace, T', found by the accelerated verification algorithm. The correspondence preserves the clock valuation on the state immediately following the loop, ensuring that the results of verification are preserved.

Lemma 1 (Loop acceleration equivalence). *For any sub-trace,*

$$\begin{aligned}&\langle l_{pre}, C_{\mathrm{pre}}\rangle \rightarrow\\ &\qquad\Big(\langle l_{top}, C_0\rangle \xrightarrow{\delta} \langle l_{top}, C_1\rangle \rightarrow\\ &\qquad\Big(\langle l, C\rangle \xrightarrow{\delta} \langle l, C'\rangle \rightarrow\Big)^*\Big)\\ &\qquad\langle l_{\mathrm{post}}, C_{post}\rangle\end{aligned}$$

found by the standard verification algorithm, there exists a corresponding path that will be found by the accelerated verification algorithm:

$$\begin{aligned}&\langle l_{\mathrm{pre}}, C_{\mathrm{pre}}\rangle \rightarrow\\ &\qquad\Big(\langle l_{top}, C_0\rangle \xrightarrow{\delta} \langle l_{\mathrm{top}}, C_1\rangle \rightarrow\\ &\qquad\Big[\langle l_{bot}, C\rangle \xrightarrow{\delta} \langle l_{bot}, C'\rangle \rightarrow\\ &\qquad\Big(\langle l_{top}, C''\rangle \xrightarrow{\delta} \langle l_{top}, C'''\rangle \rightarrow\Big)?\Big]?\Big)\\ &\qquad\langle l_{post}, C_{post}\rangle\end{aligned}$$

That is, the trace from the accelerated loop enters the location immediately outside the loop with the same clock valuation as the un-accelerated sub-trace. Here "" is Kleene star, "?" is zero or one, $l \in \{l_{top}, l_{bot}\}$ and $l_{\mathrm{post}} \notin \{l_{top}, l_{bot}\}$.*

Less formally, for every path through a loop in the original model, there is a corresponding trace in the accelerated model that ends in the same state.

Proof (Iff).

Base model path implies accelerated path: Assume that there exists an exit state, $\langle l_{\mathrm{post}}, C_{\mathrm{post}}\rangle$, generated by the original model that is *not* generated by the accelerated model.
There are two cases:

Case 1. The path enters l_{top} and then exits the loop on the next discrete transition. In this case the proof is immediate, since the loop acceleration procedure does not change the path at all.

Case 2. The path goes at least once through the loop before exiting. In that case there must be some state in the loop, $\langle l, C\rangle$ for $l = l_{\mathrm{top}}$ or $l = l_{\mathrm{bot}}$, and t a discrete transition, such that $\langle l, C\rangle \xrightarrow{t} \langle l_{\mathrm{post}}, C_{post}\rangle$ and C is not contained in the DBM of the corresponding accelerated search state, but C is contained in the DBM of a standard search state. That follows because the transition t is not changed by the loop acceleration: if the destination state is not reachable,

and the transitions are the same, the predecessor state must be unreachable.
This is a contradiction, because the accelerated DBMs are relaxations of the corresponding base model DBMs, since the loop acceleration only removes constraints. It follows from this that any clock states contained in the zones of the base model verifier will be contained in the zones of the accelerated model.

Accelerated path implies base model path: As in the previous direction, we may ignore paths that enter only l_{top} and leave the loop immediately.
So we assume that there is a path that enters both states at least once, and then reaches $\langle l_{\text{post}}, C_{\text{post}}\rangle$. So there must be $\langle l, C\rangle$ for $l = l_{\text{top}}$ or $l = l_{\text{bot}}$, and t a discrete transition, such that $\langle l, C\rangle \xrightarrow{t} \langle l_{post}, C_{post}\rangle$, and C is contained in the DBMs of only the accelerated model, and not of the base model checker.
There are two ways this could happen, corresponding to the two modifications made by Algorithm 3: either there are two clocks, c and c' desynchronized by line 6 that should not be desynchronized; or there is some clock, c, whose value is too high: the removal of upper bounds (lines 8-11) is incorrect.

Case 1 (Incorrect desynchronization). Without loss of generality, we assume that clock c is reset by the loop edge, and clock c' is not, and in the accelerated state, $c - c'$ is higher than is permitted by the base model. This cannot happen, because lower bounds on the transition times cannot prevent the clocks from diverging from each other: as the distance between transitions grows, the distance between clocks reset and clocks not reset only grows. Upper bounds on the transition times in the CIRCA model, which can keep the clocks from diverging, are not relaxed by the loop acceleration operations.

Case 2 (Clock value too high). In this case, there must be some clock, c, *not* reset by the loop, whose value is too high. But this cannot happen, since any clock that is not reset by the loop will grow monotonically as we go around the loop. Any bounds on such a clock must be imposed by urgent transitions out of the loop. But such urgent transitions impart their constraints as invariants on the locations, and loop acceleration does not relax invariants.

Theorem 1 (Loop acceleration preserves verification results). *For all traces, t explored by the standard verification algorithm, either (a) there is a corresponding trace in the loop-accelerated verification algorithm, according to Lemma 1, (b) the trace is also generated by the loop-accelerated algorithm, or (c) the trace does not reach the failure state.*

Proof. We may partition the set of traces into three subsets:

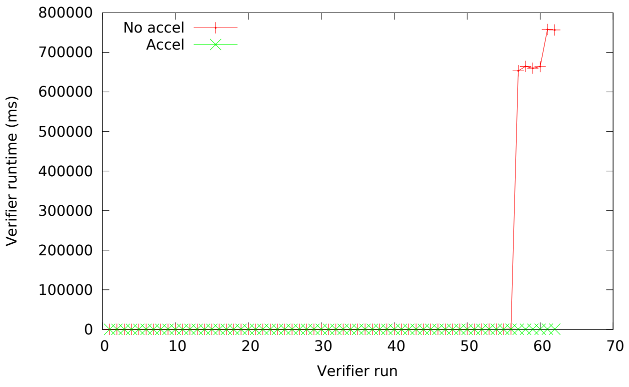


Fig. 7. The first 56 of 62 verifier runs in this single-threat domain are fairly quick even without loop acceleration. The final six runs illustrate long-duration reaction loops that cause the original verifier to perform very badly; the loop-accelerated verifier continues to perform well.

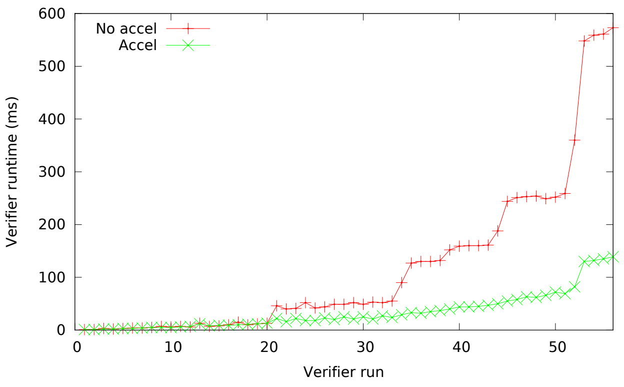


Fig. 8. Zooming in on the data in Figure 7, we can see that even in the first 56 verifier runs, the loop-accelerated verifier outperforms the original.

1. Traces that never enter an accelerated loop;
2. Traces that enter an accelerated loop and never leave that loop;
3. Traces that enter an accelerated loop and depart.

Traces of type (1) will be generated identically by the loop-accelerated verifier, so fall into case (b). Traces of type (2) cannot enter a failure state, because of the structure of CIRCA models; ergo they fall into case (c). Traces of type (3) have corresponding loop-accelerated traces, by Lemma 1.

Having shown that the loop acceleration procedure is sound, we proceed to demonstrate its utility in the next section.

10 Experimental Results

Our initial evaluation of the loop acceleration technique indicates that it can provide tremendous benefit to the

CIRCA verification system, and rarely incurs any cost that makes it worse than the baseline verification approach. For example, the graphs in Figure 7 and Figure 8 show the significant improvement in verification time in the course of planning a controller for a representative problem domain. The planning domain models a spacecraft in which a solar observing spacecraft stationed at the L1 Lagrange point must expose a sensitive instrument to possible damage in order to achieve its science goals. The planner searches for a controller that will safe the instrument before a possible solar flare can damage it. Although flares can be detected when they occur, the planner must analyze the race between the `complete_observation` transition and the `flare_damages_instrument` transition to decide whether it is necessary to power off the instrument immediately or it can wait until the observation has been completed. This creates a "reaction loop" as described in Section 7.[4]

The graphs in Figure 7 and Figure 8 show the time required by each call to the verification subroutine (*i.e.*, step 4 of Algorithm 1). Each time the planner chooses a controlled action for a new state, it invokes the verifier to ensure that failure is not reachable in the current partial plan. In this case, CIRCA calls the verifier 62 times in the course of planning. Every partial plan is safe. So, the plan becomes steadily larger and the verification problem becomes increasingly difficult. However, the planner's state space size alone does not determine verification time. As we have seen, plan loops can cause an explosion in the verifier's state space. As shown in Figure 7, the verification problem becomes qualitatively more difficult at the 57th verification. Without loop acceleration, the verifier state space grows from 2145 states to 102,329 states, and the time to verify increases by a factor of 1140 (from 573 to 653,303 milliseconds). The verifier with loop acceleration recognizes the loop, so its runtime grows by less than a factor of two (increasing from 139 to 253 milliseconds). It is significant that the runtime of the unaccelerated verifier depends on the duration of the slowest temporal bounding the loop. In the accelerated verifier, the runtime is unaffected by the duration of any transition involved in the loop. Overall, the loop-accelerated planning system builds a verified plan in a total of 3.6 seconds, while the original system requires over 4000 seconds.[5]

In domains where the planner does make poor choices, creating partial plans in which failure is reachable, the loop acceleration can provide another huge benefit: the counterexamples it produces (necessary to guide backtracking as described in step 5 of Algorithm 1) can be dramatically shorter than those produced by the non-accelerated verifier. For example, in a variation of the solar observer domain with an additional possible failure condition, the planner makes some action choices that create partial plans in which failure is reachable. After the fifth action choice, the partial plan allows for the instrument to be exposed long enough to a flare to damage the instrument, which is considered a domain failure. The verification of this partial plan using the non-accelerated verifier required 164 seconds, expanded over 20,000 states, and returned a counterexample of 20,002 states. Of the 20,002 states in the counterexample, the same two state cycle repeated 10,000 times. In contrast, the loop accelerated version completed the whole planning problem in less than a tenth of a second and, on the same fifth verification, its counterexample was just six states long.

[4] Source for the evaluation domain is available at `http://www.musliner.com/david/papers/sttt2013loopacceldata.html`.

[5] Timing information was obtained by running on a Linux computer with an Intel Duo2 dual core 2.4GHz CPU (4800 BogoMips) and 6 gigabytes of memory.

11 Conclusions

We have described a new loop acceleration technique that can dramatically speed up timed automata verification. While we have developed this technique in the context of CIRCA planning, it is quite general, because CIRCA creates practical discrete controllers that are modeled by common timed automata forms, and resemble the controllers that occur in other practical domains. Our technique applies to all timed automata safety verification problems that match the single-threat two-state loop pattern we have described.[6]

Our work in this area is ongoing. We are working to generalize the loop acceleration technique to applicability beyond simple two-state reaction loops. Although our current technique has substantially expanded the set of problems that CIRCA can solve, by speeding up a common pattern, we would like to extend its applicability to problems where instead of simple loops of states, there are "meshes" of states created from multiple threats interacting with the system, potentially interleaving.

Acknowledgments

This article was supported by Office of Naval Research contract N0014-10-1-0188 via Carnegie Mellon University subaward number 1140185-240250 and Air Force Office of Scientific Research contract FA9550-12-1-0146 via Carnegie Mellon University subaward number 1150105-284227. This paper does not represent the official position or opinions of the Office of Naval Research, the Air Force Office of Scientific Research, or Carnegie Mellon University. Thanks to Ed Koeller for collecting supporting images and data. Thanks to our reviewers for many helpful questions, suggestions, and corrections.

[6] Note that, while we use the term "threat," the pattern is actually more general — any case where the controller must service an outside process that is time-pressured exhibits the same pattern.

References

1. R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.
2. K. G. Larsen, P. Pettersson, and W. Yi, "Model-checking for real-time systems," in *Proc. of Fundamentals of Computation Theory*, no. 965 in Lecture Notes in Computer Science, pp. 62–88, Aug. 1995.
3. G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal implementation secrets," in *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.
4. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004* (M. Bernardo and F. Corradini, eds.), no. 3185 in LNCS, pp. 200–236, Springer–Verlag, September 2004.
5. S. Yovine, "Kronos: A verification tool for real-time sytems," *Springer International Journal of Software Tools for Technology Transfer*, vol. 1, October 1997.
6. T. S. Hune, "Modeling a language for embedded systems in timed automata," Research Series RS-00-17, BRICS, Department of Computer Science, University of Aarhus, Aug. 2000. 26 pp. Earlier version entitled *Modelling a Real-Time Language* appeared in Fourth International Workshop on Formal Methods for Industrial Critical Systems (FMICS99) pages 259–282.
7. T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen, "Model-checking real-time control programs - verifying LEGO MINDSTORMS systems using UPPAAL," in *In Proc. of 12th Euromicro Conference on Real-Time Systems*, pp. 147–155, IEEE Computer Society Press, 2000.
8. M. Hendriks and K. G. Larsen, "Exact acceleration of real-time model checking," *Electronic Notes in Theoretical Computer Science*, vol. 65, Apr. 2002.
9. D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: a cooperative intelligent real-time control architecture," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 6, pp. 1561–1574, 1993.
10. D. J. Musliner, E. H. Durfee, and K. G. Shin, "World modeling for the dynamic construction of real-time control plans," *Artificial Intelligence*, vol. 74, pp. 83–127, Mar. 1995.
11. D. J. Musliner, M. J. S. Pelican, R. P. Goldman, K. D. Krebsbach, and E. H. Durfee, "The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees," in *AAAI Spring Symposium on Architectures for Intelligent Theory-Based Agents*, 2008.
12. D. Kortenkamp, P. Bonasso, D. J. Musliner, M. J. S. Pelican, and J. Hostetler, "Embedding planning technology into satellite systems," in *AIAA Infotech@Aerospace*, 2011.
13. M. Hendriks, *Model Checking Timed Automata - Techniques and Applications.* PhD thesis, Institute for Programming research and Algorithmics (IPA), 2006.
14. A. Fietzke, E. Kruglov, and C. Weidenbach, "Automatic generation of inductive invariants by sup(la)," Tech. Rep. MPII2012RG1-002, Max-Planck-Institut fur Informatik, 2012.
15. F. K. . Marius Bozga, Radu Iosif, "Fast acceleration of ultimately periodic relations," Tech. Rep. TR-2010-3, Verimag Technical Report, 2010. Version: 1.
16. S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen, "Flat acceleration in symbolic model checking," in *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)* (D. A. Peled and Y.-K. Tsay, eds.), vol. 3707 of *Lecture Notes in Computer Science*, (Taipei, Taiwan), pp. 474–488, Springer, Oct. 2005.
17. S. Bardin, J. Leroux, and G. Point, "FAST extended release," in *Computer Aided Verification (CAV)* (T. Ball and R. B. Jones, eds.), vol. 4144 of *Lecture Notes in Computer Science*, pp. 63–66, Springer, 2006.
18. R. B. Salah, *On Timing Analysis Of Large Systems.* PhD thesis, Institut National Polytechnique De Grenoble, Oct. 2007.
19. E. Gat, "News from the trenches: An overview of unmanned spacecraft for AI," in *AAAI Technical Report SSS-96-04: Planning with Incomplete Information for Robot Problems* (I. Nourbakhsh, ed.), American Association for Artificial Intelligence, Mar. 1996.
20. D. J. Musliner and R. P. Goldman, "CIRCA and the Cassini Saturn orbit insertion: Solving a prepositioning problem," in *Working Notes of the NASA Workshop on Planning and Scheduling for Space*, Oct. 1997.
21. R. P. Goldman, M. J. S. Pelican, and D. J. Musliner, "Guiding planner backjumping using verifier traces," in *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling* (S. Zilberstein, J. Koehler, and S. Koenig, eds.), pp. 279–286, June 2004.
22. C. M. Potts, K. D. Krebsbach, J. T. Thayer, and D. J. Musliner, "Improving trust estimates in planning domains with rare failure events," in *AAAI Spring Symposium on Trust and Autonomous Systems*, 2013.
23. D. Kortenkamp, M. B. Hudson, S. Bell, D. J. Musliner, M. J. S. Pelican, J. Hamell, and P. Zetocha, "Embedding planning technology into satellite systems," in *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2012.
24. C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The tool kronos," in *Hybrid Systems III: Verification and Control*, pp. 208–219, 1996.
25. R. Alur, "Timed automata," Tech. Rep. MS-CIS-98-10, University of Pennsylvania, 1998.
26. R. Alur, "Timed automata," in *Working Notes of the NATO-ASI Summer School on Verification of Digital and Hybrid Systems*, 1998.
27. D. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Automatic Verification Methods for Finite State Systems* (J. Sifakis, ed.), vol. 407 of *Lecture Notes in Computer Science*, pp. 197–212, Springer Berlin / Heidelberg, 1990.
28. R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *Artificial Intelligence*, vol. 49, no. 1–3, pp. 61–95, 1991.
29. P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, pp. 81–98, Jan. 1989.