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
211 N. First St.
Minneapolis, MN 55401
{rpgoldman,mpelican,musliner}@sift.net

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 any size loop 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.

Introduction

The ability to automatically prove reachability properties of programs or plans have many applications in computer science, including safety proofs for plans generated by classical planning systems. Although intractable in the worst-case, algorithmic improvements and modern computing hardware have made these model-checking techniques practical for larger and larger problems. We describe a specific challenge to existing model-checking algorithms for timed automata and a practical solution to many instances of this challenge.

The problem arises when state transitions of greatly different temporal latencies create interacting cycles of states. This problem was first described by Hune (2000) and Iversen (Iversen et al. 2000) in the context of verifying real-time controllers for LEGO® Mindstorms™ robots. In some cases where two transition cycles apply to the same plan state, the shorter duration transition can "fragment" (Hendriks and Larsen 2002) the larger transition, creating an explosion in the number of states considered by the verifier.

CIRCA Background

CIRCA's planning system contains two main modules of interest for this paper: a Controller Synthesis Module (CSM) that reasons 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 clarify how CIRCA uses model verification in the context of automated planning and how the loop acceleration technique speeds up model verification.

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). For example, Figure 1 shows several transitions from a CIRCA problem description for controlling the Cassini spacecraft during Saturn Orbital Insertion (Gat 1996; Musliner and Goldman 1997). 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.

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. The special 'do nothing' action "no_op" can be assigned. Associated with each action is a worst case execution time, an *upper bound* on the delay before the action occurs.

Temporal (uncontrollable) transitions represent uncontrollable processes. Associated with each temporal

```
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
```

Figure 1: Example transition descriptions given to CIRCA's CSM.

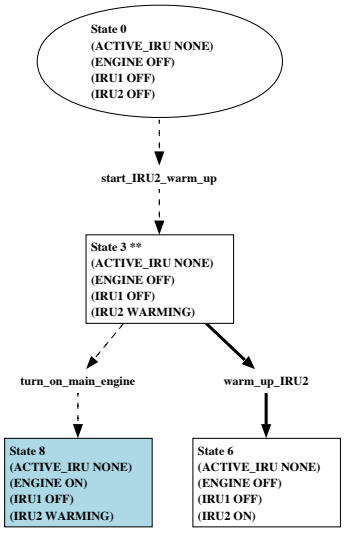

Figure 2: The beginning of a state space plan for Saturn Orbit Insertion.

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.

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 fail-*

ure 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 goto step 1 (Goldman, Pelican, and Musliner 2004).*
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.

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."[1] Unplanned states are treated as absorbing states of the system, and any verification traces that enter these states are regarded as successful. Note that this process converges to a sound and complete verification when the controller synthesis process is complete. When the verifier indicates that a controller is *unsafe*, the CSM will query it for a path to the distinguished failure state. The set of states along that path provides a set of candidate decisions to revise, as discussed in (Goldman, Pelican, and Musliner 2004).

Formal Underpinnings

In this section, we provide a mathematical description of a plan and briefly introduce the corresponding timed automaton model and algorithms used for formal safety verification.

The search described by the planning algorithm is conducted to create a plan

Definition 1 (CIRCA plan graph). *plangraph* = $\langle S, E, \vec{F}, \vec{V}, \phi, I, T, \iota, \eta, p, \pi \rangle$ *where*

1. S *is a set of states.*
2. E *is a set of edges.*
3. $\vec{F} = [f_0 ... f_m]$ *is a vector of features (in a purely propositional domain, these will be propositions).*
4. $\vec{V} = [\mathcal{V}_0 ... \mathcal{V}_m]$ *is a corresponding vector of sets of values* ($\mathcal{V}_i = \{v_{i0} ... v_{ik_i}\}$) *that each feature can take on.*
5. $\phi : S \mapsto \vec{V}$ *is a function mapping from states to unique vectors of value assignments.*
6. $I \subset S$ *is a distinguished subset of initial states.*
7. $T = U \cup A$ *is the set of transitions, made up of an uncontrollable* (U) *subset, the temporals and reliable temporals, and a controllable* (A) *subset, the actions. Each transition,* t*, has an associated delay* (Δ_t) *lower and upper bound:* $lb(\Delta_t)$ *and* $ub(\Delta_t)$*. For temporals* $ub(\Delta_t) = \infty$*, for events* $lb(\Delta_t) = 0, ub(\Delta_t) = \infty$*.*
8. ι *is an interpretation of the edges:* $\iota : E \mapsto T$*.*
9. $\eta : S \mapsto 2^T$ *is the* enabled *relationship — the set of transitions enabled in a particular state.*
10. $p : S \mapsto A \cup \epsilon$ *(where* ϵ *is the "action" of doing nothing) is the actions that the CSM has planned. Note that* p *will generally be a partial function.*
11. $\pi : S \mapsto 2^U$ *is a set of preemptions the CSM expects.*

In order to verify a partial CSM SSP graph, $\mathcal{P}$, we translate it into a timed automaton (TA) model, $\theta(\mathcal{P})$. $\theta(\mathcal{P})$ is the product of a number of individual automata.

Definition 2 (Timed Automaton (Alur and Dill 1994)). *A timed automaton A is a tuple* $\langle \mathcal{S}, s^i, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I} \rangle$ *where* $\mathcal{S}$ *is a finite set of locations;* s^i *is the initial location;* $\mathcal{X}$ *is a finite set of clocks;* $\mathcal{L}$ *is a finite set of labels;* $\mathcal{E}$ *is a finite set of edges; and* $\mathcal{I}$ *is the set of invariants. Each edge* $e \in \mathcal{E}$ *is a tuple* (s, L, ψ, ρ, s') *where* $s \in \mathcal{S}$ *is the source,* $s' \in \mathcal{S}$ *is the target,* $L \subseteq \mathcal{L}$ *are the labels,* $\psi \in \Psi_{\mathcal{X}}$ *is the* guard*, and* $\rho \subseteq \mathcal{X}$ *is a clock reset. Timing constraints* ($\Psi_{\mathcal{X}}$) *appear in guards and invariants and clock assignments. In our models, all clock constraints are of the form* $c_i \leq k$ *or* $c_i > k$ *for some clock* c_i *and integer constant* k*. Guards dictate when the model* may *follow an edge, invariants indicate when the model* must *leave a state. In our models, all clock resets re-assign the corresponding clock to zero; they are used to start and reset processes. The state of a timed automaton is a pair:* $\langle s, C \rangle$*.* $s \in \mathcal{S}$ *is a location and* $C : \mathcal{X} \rightarrow \mathbf{Q} \geq 0$ *is a clock valuation, that assigns a non-negative rational number to each clock.*

It often simplifies the representation of a complex system to treat it as a product of some number of simpler automata. The labels $\mathcal{L}$ are used to synchronize edges in different automata when creating their product.

A timed automaton *trace* is a series 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 jump 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* $0 \leq t' \leq t$*,* t' *satisfies the invariant* $I(s)$*.*

Definition 4 (Jump Transition). *A jump transition,* $\langle s_0, C \rangle \xrightarrow{e} \langle s_1, C' \rangle$*, for some* $e \in E$ *can occur when* C *satisfies the guard of* e*,* $\psi(e)$ *and* C' *satisfies the reset of* e *applied to* C*,* $\rho(e, C)$*.*

[1] This is a conservative heuristic in the sense that it is conservative about identifying search failures. Any failure that is detected in this process is definitely a true failure (soundness), but the verifier may fail to identify failures (incompleteness). At the limit, when the control program is complete, the verifier will be both sound and complete.

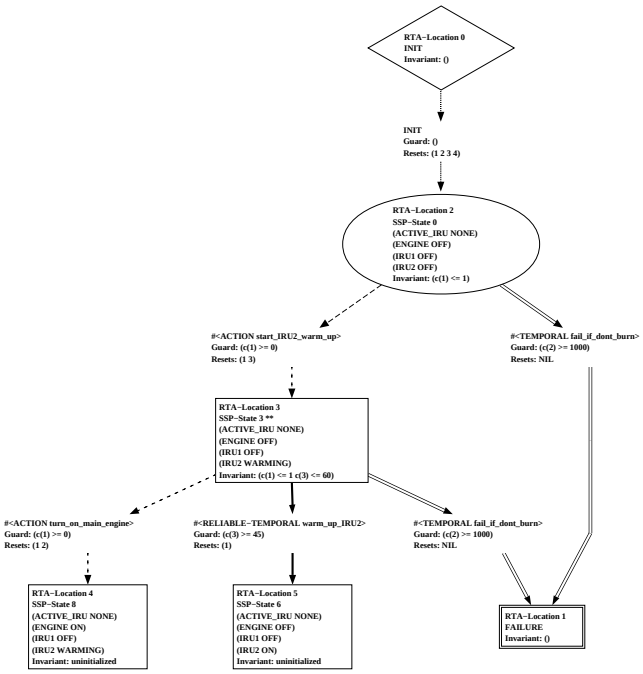

Figure 3: The timed automaton model corresponding to Figure 2.

Definition 5 (Time Quotient). *The time quotient 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 SSP graph is the time quotient of a TA model of the corresponding plan. The translation of SSP graphs to TA models is described in (Goldman, Musliner, and Pelican 2002).

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 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.

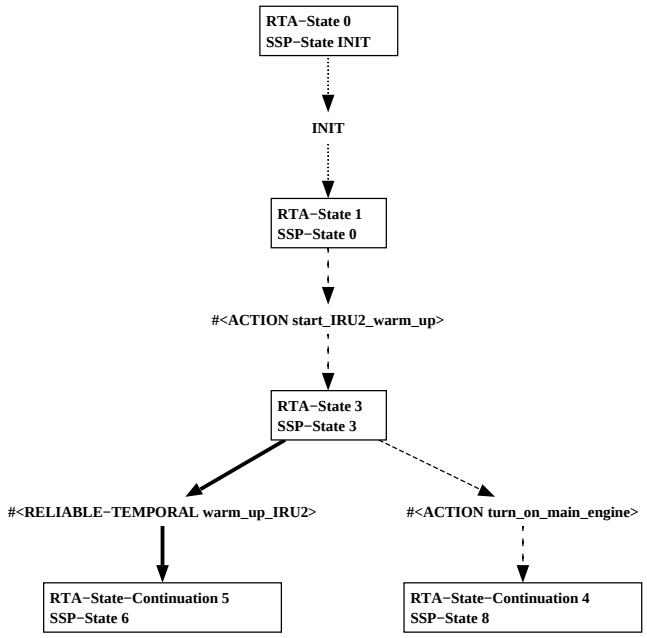

Figure 4: The timed automaton reachability space corresponding to Figure 2.

Algorithm 2 (Reachability Verification).

1. *let openlist := initial state ($\langle s^i, \mathbf{0} \rangle$)*
2. *if openlist = $\emptyset$ then return **safe**;*
3. *let state := pop(openlist);*
4. *if visited(state) then goto 2;*
5. *if failure(state) then return **unsafe**;*
6. *let succ := successors(state)*
7. *openlist := openlist $\cup$ succ;*
8. *goto 2;*

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(Alur 1998). 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 6). 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 (Goldman, Pelican, and Musliner 2004) for an explanation of our technique for using counterexample traces to

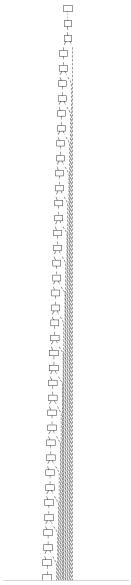

Figure 5: A portion of the fragmented verifier state space resulting from a CIRCA reaction loop.

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, we modify Algorithm 2 by adding a step after 4 as follows:

4.5 if not action-assigned(state) then goto 2;

Note that when 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.

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 (2002) 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). Figure 5 gives a qualitative impression of what can happen in this kind of situation.

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 towards a position at which it can carry out some task. 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 navigating to the destination, and the need for course corrections can also recur relatively 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 considers every possible way that the clocks representing the course correction processes interact with the clocks for the navigation process and opportunity exploitation. The clock zone techniques for collapsing the temporal state space (Alur 1998) 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 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 — 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.

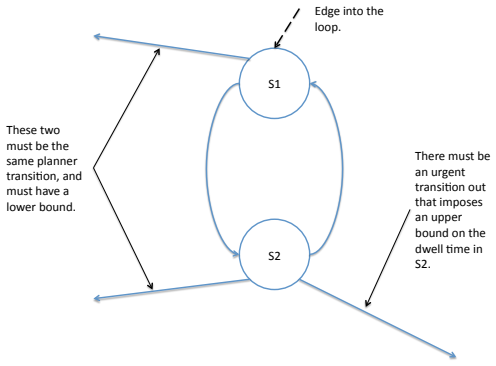

Figure 6: An illustration of the subgraph pattern that provides candidates for loop acceleration in CIRCA.

Loop acceleration for reaction loops

In the course of verification, our technique looks for candidates that meet the reaction loop pattern described above. Specifically, in the forward verification search, the verifier checks for states where:

- There is a backedge from the current location[2] to the location from which the current state was reached.
- There is a long-duration process that is active in both of the locations.
- There is a transition in the current location, *other than the backedge*, that imposes an upper bound (invariant) on the state.

To return to our earlier example, consider a looping pair of locations:

- In the first state, the vehicle is navigating smoothly. In this state, the system would like to send a transmission (imposes a deadline by a transition to an out-of-loop state where the transmission has been sent), but it may encounter an obstacle, carrying it to the second loop state.
- In the second loop state, the system will execute a course correction, that will carry it back to the first state.

In both states, the vehicle is making progress towards reaching its destination, at which it will perform some specified series of actions.

In the context of the CIRCA solver, the very large loop-related state space depicted in Figure 5 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.

Which of these possibilities happens 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 synchronized. Once we take these facts into account, we may simply remove the upper bounds on any clock that is not reset in the loop, and thus capture the exit conditions (the state of the clocks upon leaving the loop) very simply in difference bound matrices. The ability to simply desynchronize or synchronize clocks causes the state space to collapse together drastically. We may do a simple static check to verify if the state of the system *in* the loop remains safe, as well.

[2]Discrete portion of the state.

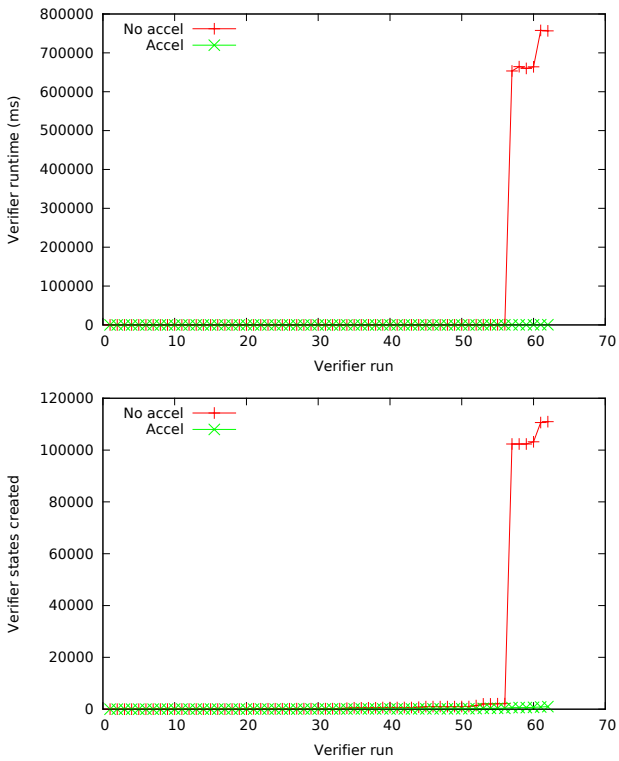

Figure 7: The first 56 of 72 verifier runs in this single-threat domain are fairly quick even without loop acceleration. The final six runs illustrate long-duration reaction loops that cause the original verifier to perform very badly; the loop-accelerated verifier continues to perform well.

The current version of CIRCA incorporates this loop acceleration technique as an option, and on many scenarios speeds up problem solving sufficiently to make previously infeasible problems solvable. In the next section, we demonstrate this with test results.

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 compare the verifier performance on 62 verification runs during planning for a spacecraft domain in which a single threat interacts with opportunities to achieve a single goal. In this domain the planner never makes an incorrect choice, so each of these verifier runs actually explores the entire reachable state space and concludes failure is not reachable. 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.

In domains where the planner does make poor choices, the loop acceleration can provide another huge benefit: the counterexamples it produces can be dramatically shorter

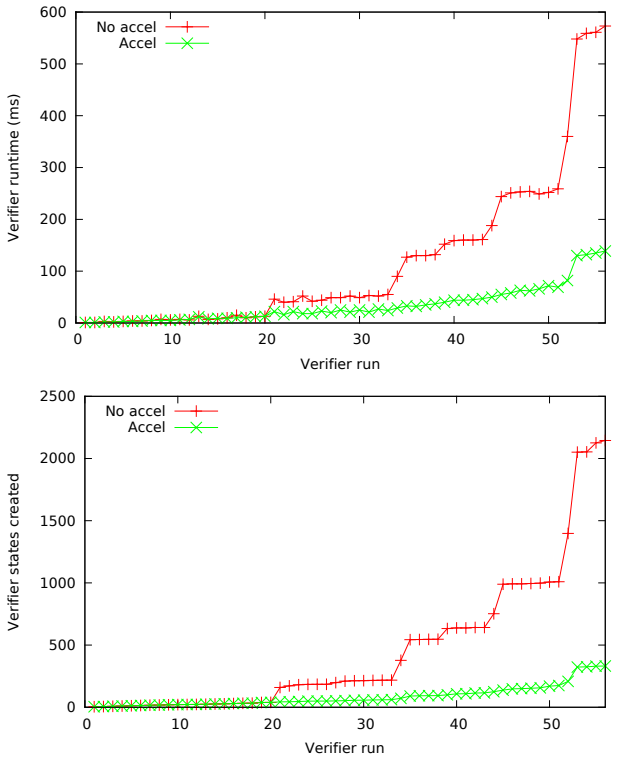

Figure 8: Zooming in on the data in Figure 7, we can see that even in the first 56 verifier runs, the loop-accelerated verifier outperforms the original.

than the non-accelerated verifier. For example, in another spacecraft planning domain with additional action choices that confuse the planner's heuristic guidance, the fifth verification run in the non-accelerated verifier required 164 seconds, expanded over 20,000 states, and returned a counterexample of 20,002 states (0-1-4-1-4...1-4-11)! 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 (0-1-4-1-4-11).

Related work

The most closely-related work to ours is the work on loop acceleration for UPPAAL done by Hendriks and Larsen (Hendriks 2006; Hendriks and Larsen 2002). 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" LTL 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.

Such a technique would not only not help us in the verification of CIRCA control plans, 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.

Conclusions

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.[3] We are also working to translate our informal proof of correctness of the transformation into one that is more formal and publishable.

Acknowledgments

This article was supported by Office of Naval Research contract N0014-10-1-0188 via Carnegie Mellon University subaward number 1140185-240250. This paper does not represent the official position or opinions of Office of Naval Research or Carnegie Mellon University.

References

Alur, R., and Dill, D. L. 1994. A theory of timed automata. *Theoretical Computer Science* 126:183–235.

Alur, R. 1998. Timed automata. In *Working Notes of the NATO-ASI Summer School on Verification of Digital and Hybrid Systems*.

Gat, E. 1996. News from the trenches: An overview of unmanned spacecraft for AI. In Nourbakhsh, I., ed., *AAAI Technical Report SSS-96-04: Planning with Incomplete Information for Robot Problems*. American Association for Artificial Intelligence.

Goldman, R. P.; Musliner, D. J.; and Pelican, M. J. S. 2002. Exploiting implicit representations in timed automaton verification for controller synthesis. In Tomlin, C. J., and Greenstreet, M. R., eds., *Hybrid Systems: Computation and Control (HSCC 2002)*, number 2289 in LNCS. Springer Verlag. 225–238.

Goldman, R. P.; Pelican, M. J. S.; and Musliner, D. J. 2004. Guiding planner backjumping using verifier traces. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, 279–286.

Hendriks, M., and Larsen, K. G. 2002. Exact acceleration of real-time model checking. *Electronic Notes in Theoretical Computer Science* 65(6).

Hendriks, M. 2006. *Model Checking Timed Automata - Techniques and Applications*. Ph.D. Dissertation, Institute for Programming research and Algorithmics (IPA).

Hune, T. S. 2000. Modeling a language for embedded systems in timed automata. Research Series RS-00-17, BRICS, Department of Computer Science, University of Aarhus. 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.

Iversen, T. K.; Kristoffersen, K. J.; Larsen, K. G.; Laursen, M.; Madsen, R. G.; Mortensen, S. K.; Pettersson, P.; and Thomasen, C. B. 2000. Model-checking real-time control programs - verifying lego mindstorms systems using uppaal. In *In Proc. of 12th Euromicro Conference on Real-Time Systems*, 147–155. IEEE Computer Society Press. also available as RS-99-53.

Musliner, D. J., and Goldman, R. P. 1997. CIRCA and the Cassini Saturn orbit insertion: Solving a prepositioning problem. In *Working Notes of the NASA Workshop on Planning and Scheduling for Space*.

[3] Note that where we say "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.