Appears in Proc. Int'l Conf. on Intelligent Robots and Systems (IROS)
Maui, Hawaii, 2001

Planning with Increasingly Complex Executive Models

David J. Musliner, Robert P. Goldman, Michael J. S. Pelican
Automated Reasoning Group
Honeywell Laboratories
3660 Technology Drive
Minneapolis, MN 55418
{musliner, goldman, pelican}@htc.honeywell.com

Abstract

We are developing autonomous control systems for mission-critical domains that require hard real-time performance guarantees. To automatically build reactive plans that meet these requirements, we use formal verification (model checking) techniques to assess the quality of plans as they are built. The verification process uses precise *timed automaton* models of the executive that will run the resulting reactive plan. This reflexive modeling allows our system to formally verify not just that its plans are correct, but that they will be executed correctly.

1 Introduction

In hazardous or mission-critical domains such as flying aircraft and controlling industrial plants, the emphasis on safety and predictability makes it difficult to deploy intelligent or self-adaptive control systems. In addition to rigorous performance requirements, these types of applications require fine-grained certification and extremely high levels of confidence from their human designers and funders.

1.1 Executives Considered Harmful

Most recent work in intelligent autonomous control has emphasized increasingly complex plan executives. These complex executives (e.g., RAPs [4], PRS [6], Remote Agent Executive [9]) support task decomposition, action sequencing, persistent goals, and powerful world modeling capabilities. They provide specialized, very powerful and concise programming languages. The advantage is that engineers can directly encode their knowledge about task decomposition and interactions, and higher-level planners can build relatively simple plans that rely on the executive to handle action failures and other forms of uncertainty. The disadvantage is that, because so much detail has been removed from the plan and left implicit in the executive, it is not clear that the plan will be executed as expected.

1.2 Simplify!

We take the position that the best way to achieve reliable, robust, and trustworthy autonomy is through the use of a predictable executive supporting simple execution semantics that can be directly considered by a planner. That is, a system in which the planner generates plans satisfying certain verifiable properties (e.g., timeliness and correctness), which are then predictably and reliably executed. Ours is essentially a high-level, control-specific, automatic programming paradigm. The system designer provides a description of primitive sensing and control actions, a description of the domain and its dynamics, and a description of the system's goals. The system generates and executes a plan composed of primitives and combination functions (control logic) to reliably achieve the goals. Once the system code itself has been certified, the only further verification/certification requirements apply to the input models of primitives and the domain; each plan (program/controller) is itself verified automatically during generation.

CIRCA is an autonomous, self-adaptive control architecture designed specifically for mission-critical domains. CIRCA combines on-line planning and scheduling systems with a very simple, very predictable real-time plan executive. To increase designer confidence and ensure that the plans that CIRCA makes are well-founded, we use formal verification techniques in the planning process. CIRCA dynamically creates time-constrained reactive control plans (cyclic loops of Test-Action Pairs, or TAPs) based on its expectations about future world states and its own potential actions. CIRCA raises the timeliness aspects of plan execution to the same level of concern as the logical correctness standards associated with traditional planning. This involves not just reasoning about time at a coarse level during plan generation, but also detailed timing information that explicitly accounts for sensing activity, the delays between sensing and action [8], communication delays, and the lowest-level details of action selection and execution.

This paper describes how CIRCA uses an explicit, formal model of its plan executive during the planning process to verify that the plans it is constructing will be executed in a timely and correct fashion. Using this self-modeling, CIRCA is able to predict and avoid several types of undesirable behaviors that may result from a less-rigorous integration of planning and execution models.

In fact, CIRCA employs three increasingly complex models of plan execution at different times during the planning process. The models have different levels of precision and information needs, and correspondingly different computational costs.

1. **Most Abstract: Simple Timing Estimates** — When making a decision about which of several possible actions to plan for a state, CIRCA uses its simplest model of execution: actions are assumed to take only their worst-case execution times, and are considered completely independently. That is, CIRCA does not worry about what other actions have been planned, or how quickly they must be performed, etc. This level of execution modeling is similar to a traditional planner, except that CIRCA is concerned about metric timing information and non-volitional (uncontrollable) events that traditional planners cannot handle.
2. **Less Abstract: Verify Independent TAPs** — After each action decision is made, the CSM uses its formal verification system to confirm that the planner's estimates of timing relationships are correct. This verification process uses models of planned actions that are considerably more accurate than method 1 above. This paper is focused on describing this level of modeling.
3. **No abstraction: Verify TAP Schedule** — The third form of modeling and verification takes place after an entire plan has been synthesized and it is thought to be correct (preventing failure) and desirable (achieving goals). By "thought to be correct," we mean checked using the more abstract models described above in (1) and (2). When this test has been passed, the CSM builds an executable TAP schedule and performs a final verification process. With the TAP schedule available, the final verification process is even more accurate than method 2 above. The final verification can consider the order in which TAPs are run, the actual TAP test expression costs (which can vary non-monotonically as the plan is being formed and actions are planned for more than one state), as well as the various internal overhead delays associated with the real executive. We are currently testing the first implementation of this final verification process.

2 The Controller Synthesis Module

CIRCA's CSM automatically synthesizes real-time reactive controllers that guarantee system safety when run on CIRCA's real-time subsystem. The CSM takes in a description of the processes in the system's environment, represented as a set of time-constrained transitions that modify world features. These transition descriptions are similar to STRIPS operators with the addition of timing information and nondeterministic outcomes. For example, Figure 1 shows several transitions taken from a problem where CIRCA is to control the Cassini spacecraft in Saturn Orbital Insertion.[1]

The CSM reasons about transitions of three types:

Action transitions represent actions performed by the RTS. These parallel the operators of a conventional planning system. Associated with each action is a worst case execution time, an *upper bound* on the delay ($\Delta(a) \leq t$) before the action occurs.

Temporal transitions represent uncontrollable processes, some of which may need to be preempted. Associated with each temporal transition is a *lower bound* on its delay ($\Delta(\mathrm{tt}) \geq t$). Transitions that have a delay lower bound of zero are referred to as "events," and are handled specially for efficiency reasons.

Reliable temporal transitions represent continuous processes that may need to be employed by the CIRCA agent. For example, when CIRCA turns on an IRU it initiates the process of warming up that equipment; the process will complete after some delay. Reliable temporal transitions have both upper and lower bounds on their delays.

2.1 CSM Algorithm

Given problem representations as above, the controller synthesis (planning) problem can be posed as *choosing a control action for each reachable state (feature-value assignment) of the system*. This problem is not as simple as it sounds, because the set of reachable states is not a given — by the choice of control actions, the CSM can render some states (un)reachable.

Indeed, since the CSM focuses on generating *safe* controllers, a critical issue is making failure states unreachable. In controller synthesis, this is done by the process we refer to as *preemption*. A transition t is preempted in a state s iff some other transition t' from s must occur before t could possibly occur. The CSM achieves preemption by choosing a control action that

[1]This example is adapted from Erann Gat's "From the Trenches" [5].

```
;; Turning on an Inertial Reference Unit (IRU)
ACTION start_IRU1_warm_up
   PRECONDITIONS: '((IRU1 off))
   POSTCONDITIONS: '((IRU1 warming))
   DELAY: <= 1

;; the process of the IRU warming
RELIABLE-TEMPORAL warm_up_IRU1
   PRECONDITIONS: '((IRU1 warming))
   POSTCONDITIONS: '((IRU1 on))
   DELAY: [45 90]

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

;; if the engine is burning while the active
;; IRU breaks, we must quickly fix problem before
;; the spacecraft gets 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 planner.

is fast enough that it is guaranteed to occur before the transition to be preempted.

At the highest level of abstraction, the controller synthesis algorithm is as follows:

1. Choose a state from the set of reachable states (at the start of controller synthesis, only the initial state(s) is(are) reachable).
2. For each uncontrollable transition enabled in this state, choose whether or not to preempt it (any transition that leads to a failure state *must* be preempted).
3. Choose a control action or `no-op` for that state.
4. Invoke the verifier to confirm that the (partial) controller is safe.
5. If the controller is *not* safe, use information from the verifier to direct backtracking.
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 been chosen), terminate successfully.
8. If some unplanned reachable states remain, loop to step 1.

Figure 2 provides a simple example of the process of controller synthesis. Initially (i), there is only one state reachable, the initial (oval) state. In (ii), the CSM has chosen a control action (dashed line) for the initial state (planned states are shaded gray), that will carry the system to a goal state, *s1* (goal states are indicated by bold outlines). There is also a temporal transition (double line) that may carry the system to *s2*. In (iii), we see the CSM decide to assign `no-op` as the control action for *s1*. This is permissible because *s1* is a safe state (there are no transitions to failure from that state), and is desirable because *s1* is a goal state. In (iv), the CSM attempts to complete the controller synthesis process by assigning an action to *s2* that will carry the system to *s3*. However, this action does *not* preempt the transition from *s2* to the failure state (black). This triggers a backtrack (v), and the system chooses an alternative action for *s2* (vi) that will carry the system to *s1*. This alternative action *does* preempt the transition to the failure state, so the controller is safe. All reachable states have been planned for, so the controller synthesis process has terminated successfully.

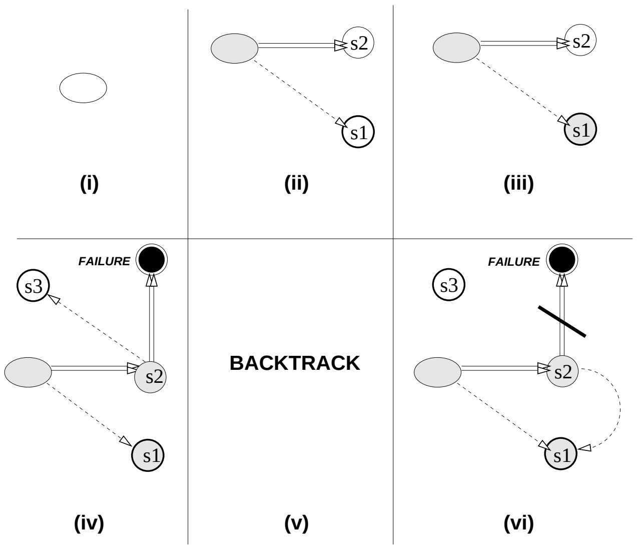

Figure 2: A simple example of controller synthesis.

During the course of the controller synthesis run above, the CSM will have employed the verifier module after each assignment of a control action (i.e., after ii, iii, iv and vi). However, at stages ii, iii and iv, the controller is not complete. At such points we use the verifier as a conservative heuristic by treating all unplanned states (e.g., *s2* in iii) 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. When the verifier indicates that a CSM-generated 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.

3 Modeling for Verification

The temporal model underlying the CSM plan graphs is deceptively complex because it is non-Markovian. We do not include time in the CSM state description (we discuss the rationale for this design decision later).

Verifying that a plan is correct requires a path-dependent computation to determine how much time remains on a transition's delay when it applies to two or more connected states. E.g., when IRU1 has failed and the system progresses through several transitions while `fail_if_burn_with_broken_IRU1` continues to apply. To complicate matters further, we cannot assume that the planned actions are completely independent: they will be executed by a real executive with limited abilities to sense and react to the world, so the planned actions will compete for this bounded reactivity.

To efficiently reason about the timing in this world model *and account for the executive's bounded reactivity*, the CSM relies on an automatic verification system. The verifier ensures that the controllers that the CSM builds are safe. When making action decisions, the CSM uses very simple reasoning, non-path-dependent to make "guesses" about transition preemptions (the only really important temporal issue in these plans). Then each of these guesses is formally verified using a faithful model of the RTS.

3.1 Execution Semantics

The controllers of the CIRCA RTS are not arbitrary pieces of software; they are intentionally very limited in their computational power.[2] The controller generated by the CSM is compiled into a set of *Test-Action Pairs* (TAPs) to be run by the RTS. Each TAP has a boolean test expression that distinguishes between states where a particular action is and is not to be executed. The test expression is a function of the plan as a whole, because the same action may be assigned to more than one state. A sample TAP for the Saturn Orbit Insertion domain is given in Figure 3.

The set of TAPs that make up a controller are assembled into a loop and scheduled to meet all the TAP deadlines. The deadlines are computed from the delays of the transitions that the control actions must preempt.If scheduling does not succeed, the CSM will backtrack to revise the controller, generating and scheduling a new set of TAPs.

3.2 Timed Automata

Now that we have a sense of the execution semantics of CIRCA's RTS, we briefly review the modeling formalism, timed automata, before presenting the model

[2]These limitations serve to make controller synthesis computationally efficient and make it simpler to build an RTS providing timing guarantees.

```
#<TAP 2>
 Tests: (AND (IRU1 BROKEN)
            (OR (AND (ACTIVE_IRU NONE) (IRU2 ON))
                (AND (ACTIVE_IRU IRU1) (ENGINE ON))))
 Acts : select_IRU2
```

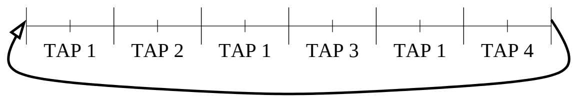

Figure 3: A sample Test-Action Pair and TAP schedule loop from the Saturn Orbit Insertion problem.

itself. A timed automaton is a finite automaton augmented with timing information.

Definition 1 (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 state;*
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 (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 $\rho : \mathcal{X} \to \mathcal{X} \cup \{0\}$ is a clock assignment [3].

Timing constraints appear in guards, invariants and clock assignments. In our modeling, all clock constraints are of the form $c_i \leq k$ or $c_i > k$ for some clock c_i and integer constant k. Informally, guards dictate when the model *may* follow an edge, invariants indicate when the model *must* leave a state, and clock assignments are used to start and reset processes.

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.

Definition 2 (Product Automaton) *Given two automata A_1 and A_2, $A_1 = \langle \mathcal{S}_1, s_1^i, \mathcal{X}_1, \mathcal{L}_1, \mathcal{E}_1, \mathcal{I}_1 \rangle$ and $A_2 = \langle \mathcal{S}_2, s_2^i, \mathcal{X}_2, \mathcal{L}_2, \mathcal{E}_2, \mathcal{I}_2 \rangle$, their product A_p is $\langle \mathcal{S}_1 \times \mathcal{S}_2, s_p^i, \mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{L}_1 \cup \mathcal{L}_2, \mathcal{E}_p, \mathcal{I}_p \rangle$, where $s_p^i = (s_1^i, s_2^i)$ and $\mathcal{I}(s_1, s_2) = \mathcal{I}(s_1) \wedge \mathcal{I}(s_2)$. The edges are defined by:*

1. *for $l \in \mathcal{L}_1 \cap \mathcal{L}_2$, for every $\langle s_1, l, \psi_1, \rho_1, s_1' \rangle \in \mathcal{E}_1$, and $\langle s_2, l, \psi_2, \rho_2, s_2' \rangle \in \mathcal{E}_2$, $\mathcal{E}_p$ contains $\langle (s_1, s_2), a, \psi_1 \cup \psi_2, \rho_1 \cup \rho_2, (s_1', s_2') \rangle$.*
2. *for $l \in \mathcal{L}_1 \setminus \mathcal{L}_2$, for $\langle s_1, l, \psi_1, \rho_1, s_1' \rangle \in \mathcal{E}_1$ and $s_2 \in \mathcal{S}_2$, $\mathcal{E}_p$ contains $\langle (s_1, s_2), l, \psi_1, \rho_1, (s_1', s_2) \rangle$. Likewise for $l \in \mathcal{L}_2 \setminus \mathcal{L}_1$.*

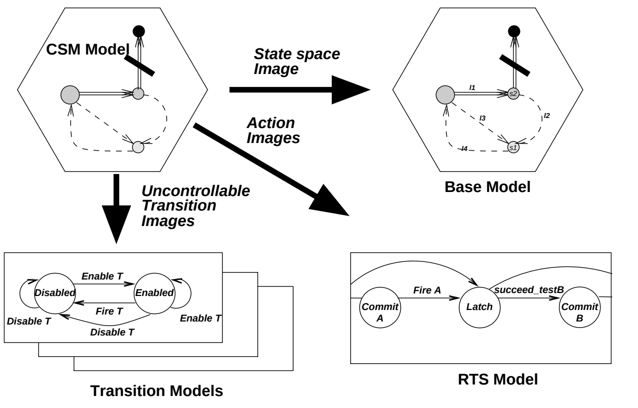

Figure 4: A pictorial summarization of the verifier model and its relation to the CSM model.

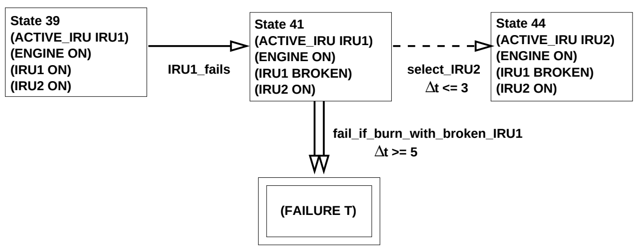

Figure 5: A portion of the CIRCA plan graph showing a simple preemptive action.

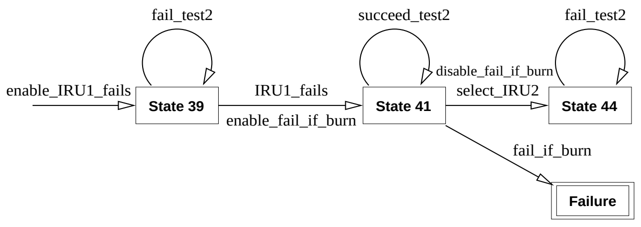

Figure 6: A portion of the verifier base model corresponding to Figure 5.

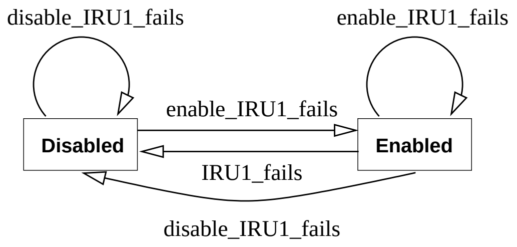

Figure 7: The verifier transition model corresponding to the event in Figure 5.

3.3 Modeling CIRCA with Timed Automata

CIRCA translates the CSM model into a set of interacting timed automata for a timed automaton verifier (see Figure 4). The use of multiple automata permits us to accurately and elegantly capture the interaction of multiple, simultaneously operating processes. The starting point of the translation is the CIRCA plan graph, constructed by the CIRCA Controller Synthesis Module. The plan graph captures all of the planned reachable states as well as the planned actions and preemption decisions discussed above. A small portion of the plan graph for our running example is shown in Figure 5. In this section of the graph, CIRCA has predicted the possibility of an IRU1 failure during an engine burn, and has planned to definitely take the `select_IRU2` action before the transition to failure could possibly occur.

As we have outlined above, the CSM's plan graph is a poor representation of the actual execution semantics of the CIRCA control system. There is not a single plant taking one action at a time; the environment is made up of a number of processes that are executing concurrently. These processes are represented as the nonvolitional transitions in a CSM domain description: the temporals, events and reliable temporals. The actions of the CIRCA control system occur concurrently with these environment processes. To model the CIRCA plan for verification, we construct several parallel timed automata that capture these separate processes.

As illustrated in Figure 4, there is one "base machine," the locations of which correspond to the states of the CSM model. For example, Figure 6 illustrates a portion of the base model corresponding to Figure 5. The base machine captures the overall state of the system and its environment. The base machine interacts with a number of "transition machines," that correspond to the transitions the CSM reasons about. This interaction is captured by the labels on the edges of the various machines; these ensure that the base machine state reflects the effect of the transitions and ensure that the state of the transition machines accurately indicate whether or not a given process is enabled in a particular system state. Note that there are no clocks or timing constraints in the base machine; all timing constraints will be handled by other automata in the composite model.

For every uncontrollable transition, there is a separate timed automaton modeling that process. Figures 7 and 8 respectively illustrate the transition models for the `IRU_fails` event and the `fail_if_burn_with_broken_IRU1` temporal transition to failure. Notice that the event transition is not constrained by any time bounds. The temporal transi-

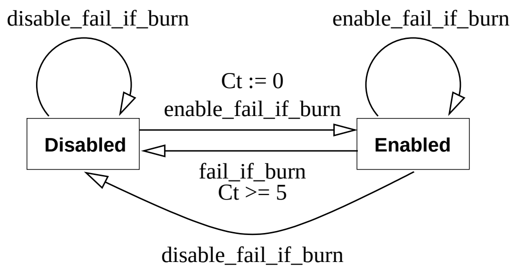

Figure 8: The verifier transition model corresponding to the temporal transition to failure in Figure 5.

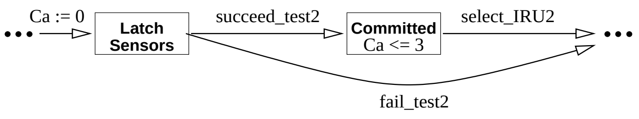

Figure 9: A portion of the verifier RTS model corresponding to the action planned in Figure 5.

tion in Figure 8, on the other hand, uses a unique clock `Ct` to ensure that the failure transition cannot occur until it has been enabled for at least 5 time ticks. Each temporal transition will have its own separate clock. Again, the edge labels synchronize the transition model with the base model, so that if this transition automaton is able to execute its `fail_if_burn` transition, the base model will recognize that a failure state is reachable.

The *RTS model* represents all the actions of the CIRCA executive in a single automaton. The RTS cycles over the TAP schedule, successively executing each TAP's test expression and, if the expression returns true, the TAP's action. The TAP test expression may be a boolean combination of primitive tests of world features. To ensure that the TAP's test expression perceives only a time-consistent view of the world, we require that the RTS latch all of the sensor values used by a TAP at the start of the TAP.

As illustrated in Figure 9, the timed automaton model of the RTS moves through a series of "latched" and "committed" states representing this execution cycle. A single clock is used to capture the timing constraints on RTS behavior. After the RTS model moves into a TAP's latch state (and the clock is reset to zero), the TAP's tests are represented as occurring by two transitions, a `succeed_` and `fail_` transition. These are synchronized with transitions in the base model that indicate whether the current world state satisfies the TAP's test expression. If so, the RTS model proceeds to a committed state, in which it definitely executes the TAP's action. The committed state has an invariant that causes the action to occur before its upper time bound. After the TAP has either fired or not, the RTS model moves on to the next segment representing the next TAP in the schedule loop.

3.4 What We Verify

There are two classes of safety violations that we look to the verifier to detect. The obvious one is a transition to the CSM's distinguished failure state. The second is a failure to successfully preempt some transition that does not carry the system directly to a failure state. These are transitions that the CSM has decided to preempt in order to make other states unreachable, possibly to make the controller smaller and more efficient or to avoid other states from which the failure state will be reachable. To detect the second class of safety violations, for each state in the base machine we add a transition to the distinguished failure state for each transition that the CSM intends to preempt.

3.5 Final Verification of Complete Controller

The above model correctly represents the interaction between the environment and the RTS. However, it is necessarily only heuristic in nature, until we have a complete plan. The reason is that until we know the entire plan, we cannot know the exact TAP tests or the shape of the TAP loop (see Figure 3), and hence cannot know the true delays before actions occur. Currently, we take the admissible, but overoptimistic, step of assuming the agent will immediately choose to take the action for its current state.

Once the controller synthesis process has completed, and the TAP loop has been generated, we run the verification algorithm one final time. At this time, we know the exact shape of the TAP loop, and hence the exact execution semantics of the plan. We do not have space here to outline the TAP loop translation process; details will be given in the full paper.

4 Related Work

Asarin *et al.* [2] developed a similar, game-theoretic method of synthesizing real-time controllers that do account for time and uncontrollable events. This work stopped at the design of the algorithm and derivation of complexity bounds; to our knowledge it was not implemented. This approach has been implemented for the special case of automatically synthesizing schedulers based on Petri Net designs [1], using the KRONOS model-checking program. Our work differs in being aimed at a different class of control problems, involving controlling devices in an active environment. We also differ in constructing purely reactive (memoryless) controllers.

Kabanza [7] has developed work very similar to ours in scope and intention. He incorporates time by effectively imposing a system-wide clock and progressing the controller one "tick" at a time. In control problems with widely varying time constants, this approach will lead to an explosion of states; we have adopted model-checking techniques that minimize this state explosion.

5 Conclusions

The CIRCA CSM is a novel application of automatic verification systems to automatic synthesis of controllers (planning). Previous attempts to use automatic verifiers in planning have limited themselves to simpler execution semantics and/or simply assumed that the plans could be implemented correctly. Our system has a rich model of the execution of its timed controller, that reflects the behavior of a hard real-time executive.

While the examples shown here are quite small, it should be clear that this translation into timed automata is highly verbose. Consider, for example, that every state of the base model must include one `succeed_` or `fail_` labeled edge for each planned TAP. Furthermore, this multi-automaton model cannot be implemented in a naive way, as this leads to a state space explosion. Our implementation uses a more efficient algorithm that exploits features of the CIRCA domain model and lazily builds the product automaton, producing speedups of two orders of magnitude on our test problems. Details of the algorithm will be presented in forthcoming publications.

Acknowledgments

This material is based upon work supported by DARPA/ITO and the Air Force Research Laboratory under Contract No. F30602-00-C-0017.

References

[1] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S.Tripakis, and S.Yovine, "A framework for scheduler synthesis," in *Proceedings of the 1999 IEEE Real-Time Systems Symposium (RTSS '99)*, Phoenix, AZ, December 1999, IEEE Computer Society Press.

[2] E. Asarin, O. Maler, and A. Pneuli, "Symbolic controller synthesis for discrete and timed systems," in *Proceedings of Hybrid Systems II*, P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, Springer Verlag, 1995.

[3] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The Tool KRONOS," in *Hybrid Systems III*, 1996.

[4] R. J. Firby, "An Investigation into Reactive Planning in Complex Domains," in *Proc. National Conf. on Artificial Intelligence*, pp. 202–206, 1987.

[5] 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, editor. American Association for Artificial Intelligence, March 1996. Available at `http://www-aig.jpl.nasa.gov/home/gat/gp.html`.

[6] M. P. Georgeff and F. F. Ingrand, "Decision-Making in an Embedded Reasoning System," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 972–978, August 1989.

[7] F. Kabanza, "On the Synthesis of Situation Control Rules under Exogenous Events," in *Theories of Action, Planning, and Robot Control: Bridging the Gap*, C. Baral, editor, number WS-96-07, pp. 86–94. AAAI Press, 1996.

[8] D. J. Musliner, E. H. Durfee, and K. G. Shin, "Predictive Sufficiency and the Use of Stored Internal State," in *Proc. AIAA/NASA Conf. on Intelligent Robots in Field, Factory, Service, and Space*, March 1994.

[9] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith, "Robust Periodic Planning and Execution for Autonomous Spacecraft," in *Fifteenth International Joint Conference on Artificial Intelligence*, 1997.