World Modeling for the Dynamic Construction of Real-Time Control Plans

David J. Musliner
Institute for Advanced Computer Studies
The University of Maryland
College Park, MD 20742
musliner@umiacs.umd.edu

Edmund H. Durfee and **Kang G. Shin**
Dept. of EE & Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{durfee,kgshin}@eecs.umich.edu

Revision : 4.1

Abbreviated title: World Modeling for Dynamic Real-Time Control Plans

ABSTRACT

As intelligent, autonomous systems are embedded in critical real-world environments, it becomes increasingly important to rigorously characterize how these systems will perform. Research in real-time computing and control has developed ways of proving that a given control system will meet the demands of an environment, but has not addressed the dynamic planning of control actions. Building an agent that can flexibly achieve its goals in changing environments requires a blending of real-time computing and AI technologies. The Cooperative Intelligent Real-time Control Architecture (CIRCA) implements this blending by executing complex AI methods and guaranteed real-time control plans on separate subsystems. We describe the formal model of agent/environment interactions that CIRCA uses to build control plans, and we show how those control plans are guaranteed to meet domain requirements. CIRCA's world model provides the information required to make real-time performance guarantees, but avoids unnecessary complexity.

To appear in AI Journal

The work reported in this paper was supported in part by the National Science Foundation under Grants IRI–9209031 and IRI–9158473, by a NSF Graduate Fellowship, by the Office of Naval Research under Grant N00014–91–J–1115, and by the Arpa/Rome Laboratory Planning Initiative (F30602-93-C-0039). The opinions, findings, and recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the funding agencies. This paper was written when David Musliner was a graduate student at the University of Michigan, Ann Arbor. Dr. Musliner is currently also affiliated with the University of Maryland Institute for Systems Research (NSF Grant NSFD CDR-88003012).

1 Introduction

Artificially intelligent agents that are constructed in the laboratory are often unsuited to real-world domains, where the pace of interactions between an agent and its changing environment may exceed the response rate of traditional AI methods. For example, an autonomous vehicle operating in the real world needs a control system that responds quickly enough to avoid collisions with obstacles or other vehicles. This requirement for timely behavior is the defining characteristic of a class of environments known as *hard real-time* domains. Hard real-time domains have deadlines by which control responses must be produced, or catastrophic failure may occur. Other common examples of hard real-time domains include nuclear power plant control, medical monitoring, and aircraft control.

Because catastrophic failure may occur if deadlines are missed, control systems for agents operating in real-time environments must not only choose appropriate actions in varied situations, they must make those action choices at appropriate times. Research in real-time systems addresses precisely this issue, by developing methods for *guaranteeing* that the reaction rate of a control system matches the rate of change in the environment. Real-time computing is *not* about building "fast" systems; it *is* about building systems that are predictably "fast enough" to act on their environments in ways that achieve their goals [23, 49]. Real-time systems researchers have developed a powerful set of tools to prove that embedded systems meet this criterion. These tools include techniques for characterizing a system's interactions with its environment through such measures as worst-case execution time, resource requirements, and deadlines. Given this type of information, mechanisms are available to predictably schedule and execute the described behaviors and to guarantee that they will meet their deadlines.

While real-time systems research addresses timeliness issues for a given set of tasks, it does not address the source of those tasks; real-time researchers assume they are given tasks that have certain performance requirements, but the motivations for those tasks and requirements are left unspecified. Traditional AI planning research, on the other hand, has characterized the interactions of an agent and its environment in terms of state spaces and operators that move through those spaces. Planning has concentrated on searching for sequences of operators (tasks) to execute in a particular situation. Thus we would like to combine the guaranteed performance methods of real-time systems with AI planning mechanisms to build a flexible, intelligent control system that can dynamically plan its own behaviors and guarantee that those behaviors will meet hard deadlines in real-time environments.

This paper describes the techniques we have developed to model agent/environment interactions, allowing a system to integrate real-time considerations into a state-based planning representation, and to reason over this representation in a uniform and consistent manner. We describe this world model in the context of the Cooperative Intelligent Real-Time Control Architecture (CIRCA) [31, 34]. As illustrated in Figure 1, CIRCA combines parallel AI and real-time control subsystems to meet the requirements of both arbitrarily complex AI algorithms and predictable real-time control responses. The AI subsystem (AIS) performs high-level reasoning about tasks and, in cooperation

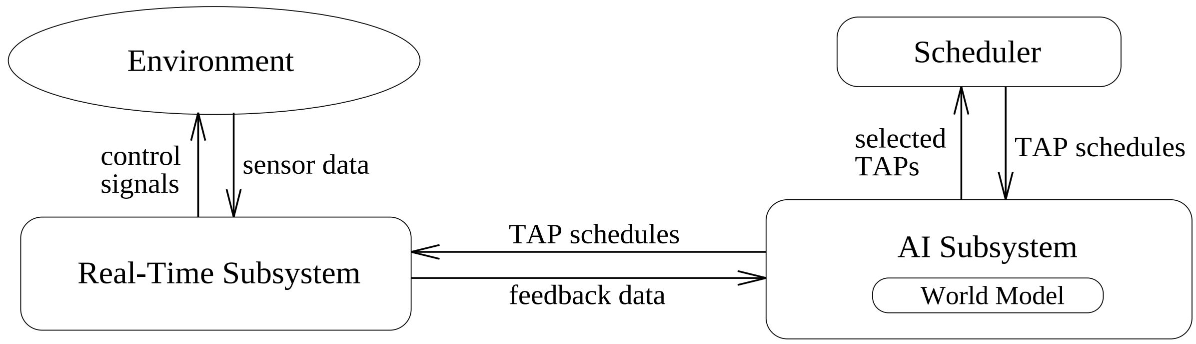

Figure 1: The Cooperative Intelligent Real-Time Control Architecture.

with the Scheduler, develops low-level control plans using low-variance primitives. These control plans are executed in a predictable, guaranteed fashion by the real-time subsystem (RTS).

CIRCA's domain-independent world modeling method provides a concise characterization of the information required to make guarantees, and allows the system to automatically derive a reactive control plan that can be guaranteed to meet the domain's deadlines and achieve the system's goals. The architecture makes a fundamental distinction between activities directed towards achieving "control-level" goals, which are guaranteed to meet domain deadlines, and behaviors for "task-level" goals, which are executed in a less-predictable manner. We will show how the world model serves not only as the basis for planning actions and making performance guarantees, but also as a key justification for the architectural division of control-level (guaranteed) and task-level (unguaranteed) goals.

By reasoning about this model of agent/environment interactions to produce its guaranteed control plans, CIRCA is also able to introspect on its own capabilities. When the system does not have sufficient resources to guarantee that it will meet all of the domain's deadlines, CIRCA can recognize this overconstraining situation from the world model, and can make explicit tradeoffs in its plans, goals, and expectations. The world model thus plays a crucial role in guiding the system's behavior and providing an underlying framework for performance guarantees.

From an AI perspective, this paper presents a method for dynamically building reactive systems based on an explicit characterization of the range of possible agent/environment interactions. From a real-time systems perspective, this paper presents a method for intelligently automating the dynamic synthesis and verification of real-time control systems.

1.1 Example Domain

We will illustrate CIRCA's world-modeling and control-planning mechanisms in the example domain shown in Figure 2. The Puma robot arm, simulated in Deneb Robotics' Igrip system, is assigned the task of packing objects (parts) arriving on the conveyor belt into the nearby box. The parts can have several shapes (e.g., square, rectangle, triangle), each of which requires a different packing strategy. The control system may not know *a priori* how to pack all of the possible types of parts. If parts of a new shape arrive, the system can stack those parts on the nearby table until it has derived an appropriate box-packing strategy. The conveyor moves at a fixed rate and the

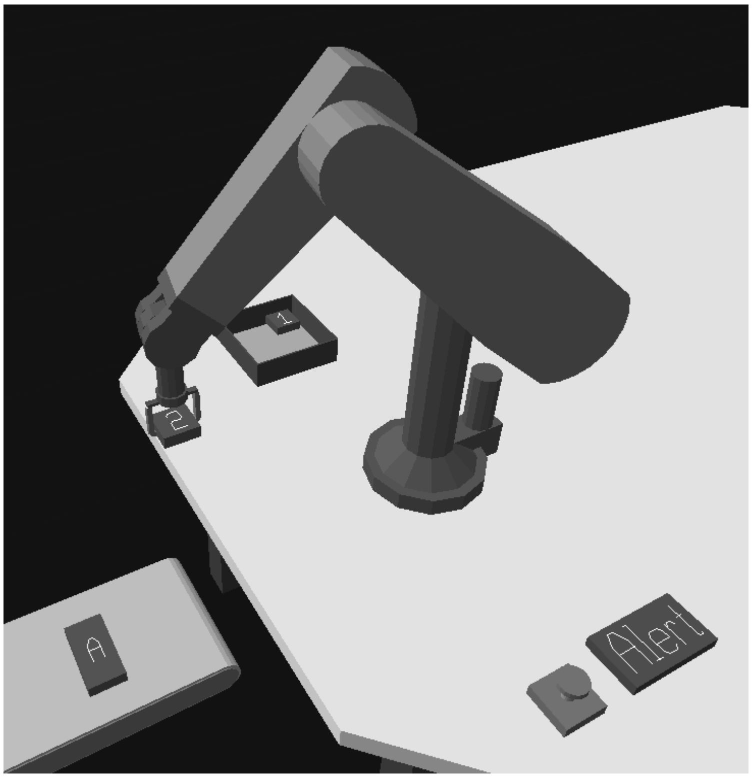

Figure 2: The example domain, in which the Puma robot packs objects from the conveyor into the box.

parts are spaced apart on the belt so that they arrive with some maximum frequency. Once at the end of the belt, each part remains motionless until the next part arrives, at which time it will be pushed off the end of the belt (unless the robot picks it up first). If a part falls off the end of the belt because the robot does not pick it up in time, the system is said to have failed.

The robot arm is also responsible for reacting to an emergency alert light. If the light comes on, the system has only a limited time to push the button next to the light, or it fails. This portion of the task represents a completely asynchronous interrupt with a hard deadline on its service time.

1.2 Organization

This paper is organized into six additional sections. In the next section, we provide more detail on the conflicting nature of AI and real-time systems, and briefly describe previous approaches to combining these methods. Section 3 presents an architectural overview of CIRCA and discusses the guarantees that the system strives to provide. Section 4 describes the world model that the system uses to build and guarantee control plans. The model was introduced briefly in [34]; this paper provides complete details on the theoretical basis for the system's guarantees. Section 5 describes how the world model is used explicitly by the system to dynamically construct and guarantee control plans. Section 6 discusses related work on modeling techniques used in the service of real-time control. Section 7 concludes with a summary of our progress and the avenues of future research we are pursuing.

2 Background: Real-Time vs. AI

AI planning research has traditionally concentrated on being able to prove that a sequence of actions will lead to a desirable state of the world. Real-time systems, however, are concerned with proving that the time needed by a set of actions will not exceed deadlines. Ideally, we would like to combine the intelligent planning methods from AI with the guaranteed performance features of real-time systems, to build an intelligent agent that could be guaranteed to succeed in its environment. But when real-time constraints are imposed on the action of planning as well as on the resultant plan, developing intelligent real-time embedded agents is very difficult.

In this section, we compare the features of real-time and AI methods. This comparison reveals a conflict between the constraints involved in making real-time guarantees and the characteristics of traditional AI methods. We briefly survey several approaches to resolving this conflict before introducing our approach in Section 3.

2.1 Real-Time Systems

As we noted in the Introduction, real-time domains are primarily characterized by deadlines. To succeed in a real-time domain, a control system must always provide required responses before their associated deadlines. Thus real-time research has focused on ways of proving that a particular set of tasks can be *guaranteed* to meet a domain's timing constraints. The most common approach to making these guarantees is to analyze the worst-case resource requirements of the set of required tasks and then build a task-execution schedule that ensures the tasks will all meet their deadlines in the worst case.

As with all such proofs, these guarantees are based on several assumptions about the nature of the tasks and domain. For example, the system must know all of the tasks to be executed, and their worst-case resource needs, before those tasks must actually be executed. Similarly, the system must know the deadlines by which the tasks must be completed, and the available execution resources. Finally, the system must be able to finish building the schedule of tasks before those tasks must actually be executed.

These characteristics of real-time systems are summarized in the second column of Table 1: a real-time system assumes an environment that may be dynamic (in the sense that the tasks required may vary at runtime) but that at least has known worst-case task requirements. Most real-time systems run numeric control algorithms with well-understood resource requirements and performance. Using these worst-case measures, it is possible to build task schedules that allocate the system's limited execution resources and provide guaranteed response times.

2.2 AI Methods

The third column of Table 1 outlines characteristics of traditional AI planning systems, and reveals a sharp contrast with real-time systems. Most AI systems are based on the "closed world" assumption: the AI-controlled agent is the only source of change in the world. Within this envi-

	Real-Time System	Traditional AI Planner	Reactive AI System	Cooperative AI & Real-Time
Environment	dynamic, known worst-case	static, closed-world, predictable	dynamic, unpredictable, unmodeled	dynamic, modeled, limited unpredictability
Tasks	classical control, numeric algorithms	search, lookahead planning	situated reaction, no lookahead	planning & reaction
Resources	limited	assumed sufficient	assumed sufficient	limited
Response Time	guaranteed	high-variance or unbounded	bounded	guaranteed reactions, high-variance planning

Table 1: Comparing features of AI and real-time systems.

ronment, the AI system's task is to plan some future course of action using projection (lookahead) and search. Most planners assume that the agent executing the plan will have essentially unlimited sensing and processing resources.

If we try to cast this type of AI method as a task within a real-time system, the fundamental problem is that planning involves searching for the solution to a generally intractable problem [5], and thus the planning process has extremely large worst-case resource requirements. The time to find a plan in the worst case may be several orders of magnitude longer than the average time to find a plan. This means that allocating resources to guarantee the worst-case response time will be very costly, and will lead to very low utilization of a system's resources [37, 46]. Furthermore, AI systems with powerful knowledge representations [5, 10] or learning abilities [9] may have unbounded worst-case response times. In these cases, it is impossible to allocate sufficient resources ahead of time, and thus real-time guarantees are not feasible.

2.3 Approaches to Real-Time AI

We can intuitively describe two fundamental approaches to integrating AI and real-time systems: a system can try to be "intelligent *in* real-time," or it can try to be "intelligent *about* real-time." In the first approach, AI mechanisms are forced to meet real-time deadlines, and the high-variance problems discussed above are relevant. In the second approach, AI methods are used to reason about the real-time tasks that must meet deadlines, but the AI process itself is not so constrained.

2.3.1 Reactive Systems

Most research on the first approach focuses on overcoming the high-variance nature of traditional planning systems, thus making it practical to embed them in real-time systems [37]. For example, "reactive" systems have been developed to rely on frequently-updated, sensor-based representations of their environment and perform little or no lookahead planning [1, 4, 11]. The features of these systems are summarized in the fourth column of Table 1. Note that, since they do not perform search, reactive systems generally have low-variance, bounded response times. While these hand-engineered systems have been quite successful, they lack a rigorous foundation supporting their capabilities. Recent research has focused on *automatically* generating reactive control systems for situated agents [15, 29, 44]. In this approach, a system is given a description of an agent's goals, its environment, and its possible actions. The system then derives a reactive control system that chooses the correct action for any particular situation within the bounds of the world model. This approach combines the logical correctness of traditional planning systems with the robust execution features of reactive systems.

However, while reactive systems may have low-variance response times, "reaction planning" systems do not yet provide real-time performance guarantees. There is no proof that the reactive systems they generate will react at a rate appropriate to the environmental changes. A system can be over-reactive if it acts prematurely, committing to a poor course of action when it has sufficient time to compute a better one. Likewise, a system can be under-reactive if it fails to keep up with changes in its environment, and its decisions are based on excessively outdated information. The effectiveness of an agent depends not on its absolute reaction speed, but on its speed *relative to its environment.* Thus a fundamental design goal for a situated agent is the ability to react to its environment at an appropriate time scale. Furthermore, since various environmental features may change at drastically different rates, an agent should be able to support reactivity on a variety of time scales, and should be able to explicitly reason about the appropriate rate of reactivity for a particular goal and environment.

2.3.2 The Cooperative Approach

The second approach to real-time AI, where an agent should be "intelligent *about* real-time," avoids embedding AI mechanisms within a real-time system. Instead, AI processing and real-time control run in parallel and are loosely coupled. In essence, the cooperative approach builds an overall system from two components: a traditional real-time system, and a real-time system designer. Real-time system designers have traditionally been people, who are given detailed characterizations of how the real-time system needs to interact with the environment. In the cooperative approach, an AI system performs the role of a system designer, explicitly allocating the system's limited execution resources for expected tasks. Because the real-time and AI components are only loosely coupled in a cooperative system, the AI processing can remain unpredictable, high-variance, and unguaranteed. This cooperative approach has motivated the design of CIRCA.

3 Overview of CIRCA

We assume that the autonomous agent our system must control will inhabit an environment in which, to survive and achieve its goals, the agent must respond actively to various types of input stimuli. Some of those responses will maintain the system's safety and some will help achieve other system goals. Within this type of environment, CIRCA is designed to make guarantees about its performance based on the fundamental restriction that the system has limited sensing, processing, and actuating resources. A direct consequence of this bounded rationality [48] and bounded reactivity [33] is that the system usually cannot simultaneously guarantee *all* the required reactions to input stimuli that may ever be required to achieve its goals. CIRCA's solution to

```
TAP place-rectangle-in-box
    :TEST (and (part-status in-gripper) (part-type rectangle))
    :ACTION (place-rectangle-in-box)
    :RESOURCES (overhead-camera arm)
    :TEST-TIME .2          [seconds]
    :ACTION-TIME 2.5       [seconds]
    :MAX-PERIOD 11.2       [seconds]
```

Figure 3: An example TAP from the robot arm domain.

this limitation has two elements. First, the system divides its overall task into subtasks that only require selected subsets of the system's possible reactions. CIRCA dynamically builds short-term control plans that are guaranteed to implement those subsets of reactions. As the agent pursues different subtasks, the appropriate reactions change, and new control plans are derived. Thus the system never tries to simultaneously implement all of the reactions required for the overall task.

CIRCA's second way of dealing with resource limitations is to gracefully degrade its guarantees. If a subtask still requires more reactive responses than can be guaranteed, the system can leave less important reactions unguaranteed. CIRCA's guarantees are based on worst-case execution times, so when guaranteed reactions use less time than they have been allotted, the system can use the remaining time to execute unguaranteed reactions. Thus CIRCA creates two classes of reactions (guaranteed and not) so that it can guarantee the timeliness of some reactions rather than none. We will discuss the value of these guarantees in Section 3.4, after presenting more details on CIRCA.

3.1 Control Plans

CIRCA's control plans take the form of cyclic schedules of simple test-action pairs (TAPs). Each TAP is essentially an annotated production rule consisting of a set of tests (or preconditions), a set of actions to take if all the tests return true, data about the sensing and actuating resources the TAP requires, and worst-case timing data on how long it takes to test the preconditions and execute the actions. During the process of building control plans (to be discussed in detail in Section 5), individual TAPs are automatically composed from primitive descriptions of actions and tests. The planning process also assigns each TAP a maximum period, fixing the longest time interval allowed between invocations of the TAP. A control plan (TAP schedule) is guaranteed to execute its component TAPs at least as frequently as their maximum periods require.

Figure 3 shows an example TAP generated automatically for our example robot task. The `TEST` specifies that the TAP is executed only if the robot has grasped the part, and knows that the part is rectangular. If these conditions are true, the robot places the part into the box. Testing and executing this TAP takes a maximum of 2.7 seconds (`TEST-TIME` + `ACTION-TIME`), and the AIS' planning process has determined that it must be run at least every 11.2 seconds (`MAX-PERIOD`) to guarantee that the current part will be processed by the time the next part arrives (thus avoiding failure).

In addition to the cyclic schedule of guaranteed TAPs, a control plan may also include a list of unguaranteed or "best-effort" TAPs. These TAPs implement reactions that are desirable, but cannot be guaranteed due to the system's bounded reactivity. If the tests of a guaranteed TAP in the cyclic schedule returns false, then an unguaranteed TAP may be executed in the time scheduled for that guaranteed TAP's action.

To facilitate our discussion, we introduce a functional notation for referencing features of a TAP τ. The function $tests(\tau)$ refers to the TAP's tests, and $actions(\tau)$ refers to the actions the TAP implements. We use $wcet(tests(\tau))$ to refer to the worst-case execution time of the TAP's tests, and likewise $wcet(actions(\tau))$ for the worst-case execution time of the TAP's actions. These are the values represented by the `TEST-TIME` and `ACTION-TIME` slots in the TAP structure. The worst-case execution time for the whole TAP is thus $wcet(\tau) = wcet(tests(\tau)) + wcet(actions(\tau))$. The best-case and actual execution times are similarly referenced by the functions $bcet(\tau)$ and $et(\tau)$. We introduce these last two notations only for the discussion in Section 4; CIRCA does not represent or reason about them.

3.2 Operations

CIRCA's operation can be viewed as a pipeline in which control plans are derived in the AIS, scheduled in the Scheduler, and then executed on the RTS. These three operations can occur simultaneously on different control plans, so that while the AIS and the Scheduler are cooperatively developing the next control plans, the RTS is executing the previous control plan and maintaining system safety. However, data flow is not strictly unidirectional through the pipeline: feedback information can flow from the RTS and Scheduler to the AIS, so that changes in the world can affect the generation of control plans. For example, the arrival of a part of an unfamiliar type will cause the RTS to temporarily stack the part on the table and notify the AIS. In response, the AIS will develop a new plan for packing the new type of part into the box.

CIRCA's primary architectural feature is the separation of real-time and non-real-time subsystems. The RTS and AIS serve different purposes within the system, and their interaction must be carefully controlled. The RTS is responsible for executing control plans in a completely predictable fashion, so that their execution matches the model used by the AIS and Scheduler. The RTS meets this criterion for TAP execution because it has no other function; it simply loops over the cyclic schedule of TAPs, testing and executing them repeatedly. Even communication in and out of the RTS is encapsulated within TAPs, so that all RTS activity is scheduled explicitly[1]. Thus control plans that make guarantees in the modeled world are executed accurately, and the model guarantees are equally valid in the real world.

The AIS and Scheduler, on the other hand, perform the complex, unpredictable reasoning required to develop guaranteed control plans, and the performance of these subsystems must not interfere with the RTS' predictable execution. To achieve this isolation, each control plan executed on the RTS is designed both to achieve a short-term goal and to ensure system safety throughout the range of environmental states that may occur during and after the accomplishment of this goal. The effect of the latter criterion, which will be explained in detail in Section 4, is to allow the

[1] For more details on the RTS, see [31].

RTS to keep the system safe while the AIS and Scheduler try to build the next control plan; the planning operation is *not* constrained to meet domain deadlines.

The planning processes of the AIS can be divided into two main levels: the planning that builds control plans (TAP schedules) to accomplish some short-term goal, and the higher-level abstraction planning that decomposes long-term goals into short-term goals for which control plans will be built. This paper is primarily concerned with the AIS planning processes that build control plans.

These control plans can implement sequential behavior, such as the series of actions required for the Puma to pick up a part from the conveyor, move to the box, and place the part in the box. Longer-term sequential behavior is achieved by downloading new control plans to the RTS. For example, if the Puma must move full boxes onto a second conveyor, the set of control reactions required for that task might form a separate TAP schedule, downloaded to the RTS when a box is filled[2]. Control is transferred to a new plan only when the system detects that it is in an acceptable state which the new plan has also been built to handle. Clearly, to implement sequential behavior in this way, the planner must decompose the task so that consecutive control plans have no hard real-time constraints between them. That is, the planner must develop consecutive control plans whose common states for transitions between the plans can be maintained or re-achieved indefinitely.

In a less-repetitive domain such as mobile robot navigation, this type of sequential activation of control plans is even more intuitive. For example, a mobile robot might be given one control plan that moves it along a hallway to a doorway, another plan to move through the doorway into a room, and another to perform some task once at a workstation in the room. These separate control plans would each use the robot's limited sensors, processors, and actuators in different ways during the different phases of operation. The system would transfer between control plans only when the mobile robot was in a safe (halted) state, so there would be no hard deadlines dictating the time by which each control plan must be built.

3.3 Control-level vs. Task-level

The dichotomy between CIRCA's real-time and non-real-time subsystems relies on the distinction between two classes of goals: control-level goals and task-level goals. CIRCA is designed to guarantee its control-level goals via the predictable execution of the RTS. Task-level goals, on the other hand, are achieved on a best-effort basis; that is, the system tries to achieve task-level goals when possible, but if time pressure or other restrictions make this impossible, the system is still considered successful. In real-time systems terminology, control-level goals correspond to hard deadlines. Frequently, control-level goals are related to system safety. For example, in our robot arm domain, the system has a control-level goal of preventing arriving parts from falling off the end of the moving conveyor belt, because parts may be fragile or explosive, and thus dropping them is considered a catastrophic failure. Task-level goals can be violated (or not achieved) without such drastic results. For example, the robot arm system is given a task-level goal to stack arriving parts in the box. However, if the emergency light goes on during that operation, it is acceptable for the

[2] This example raises the obvious possibility of caching and reusing TAP schedules— we expect that this approach could provide significant benefits, but this paper focuses on how to produce these schedules in the first place.

system to quickly place the part on the table (instead of in the box) and respond to the emergency. In this example, it is acceptable for the system to not achieve its task-level goal, and no deadline is given.

We can also conceive of task-level goals that have deadlines, but those deadlines must be "soft" or negotiable. Task-level deadlines frequently result from commitments to other agents, while control-level deadlines are derived from physical relationships between an agent and its environment. For example, a mobile robot may have a deadline for a task-level goal of arriving at some location, but missing that deadline may only require the agent to renegotiate a rendezvous with another agent at some later time. The same mobile robot, however, will have control-level goals to avoid collisions, and the actions that achieve those goals must always meet their deadlines, or the robot may be damaged. Accordingly, the system always gives priority to scheduling and guaranteeing actions that achieve control-level goals.

The distinction between task-level and control-level goals is made automatically by CIRCA, based on its analysis of the domain model, resource limitations, and prioritized goals specified by the system designer[3]. Examining this information, CIRCA can derive deadlines for the actions which achieve the various goals, and can try to maximize the number of goals it will achieve given its bounded reactivity. CIRCA may also dynamically decide that it does not have the resources required to guarantee that it will achieve all of its control-level goals. In that case, the system automatically makes performance tradeoffs which may leave some control-level goals unguaranteed, treating them essentially the same as task-level goals. Thus control-level goals are those that the system should try to guarantee, but this may not always be possible.

Linking control-level goals to system safety is a crucial concept, because it shows how the RTS and AIS can be truly isolated. Since the AIS and RTS run on separate processors, the AIS' reasoning is largely separated from the system's actual interactions with the environment. The only way that the AIS' processing affects the world (directly, not through the RTS) is in the fact that it takes up time— that is, while the AIS is building a control plan, the world "keeps going." However, even this effect can be factored out because the RTS continues interacting with the world, enforcing the guarantees on control-level goals. If those guarantees ensure the system's safety, the RTS can continue keeping the system safe for an indefinite amount of time while the AIS generates the next control plan.

CIRCA's unguaranteed TAP list provides best-effort reactions that are not guaranteed to meet any deadlines, but may run when the system has extra time available. Unguaranteed TAPs typically achieve task-level goals, and in tightly constrained circumstances they will also provide best-effort attempts to achieve control-level goals. In the degenerate case when all reactions are best-effort because the system lacks the resources needed to guarantee any, CIRCA behaves just like most other reactive systems, executing as fast as it can, with no reason to believe this speed will meet the demands of its environment. We claim that CIRCA's automatically guaranteed control performance is superior in many ways to unguaranteed control.

[3]Currently, our implementation only deals with two priorities: critical and not.

3.4 The Value of Guarantees

One main benefit of providing control-level guarantees is the *a priori* knowledge of the suitability of the control system; if CIRCA can build a guaranteed control plan, we may confidently use that plan in situations where failure is not acceptable. If CIRCA cannot provide a guaranteed control plan, this is an indication that the system does not have sufficient resources to cope with its control-level goals in the environment. In that case, CIRCA has the ability to modify its high-level plans or goals to try to build an acceptable plan. For example, the system could alter the way it decomposes a long-term goal into short-term goals, so that the timing constraints on difficult processes are relaxed. In our example domain, the system might allocate more time to the process of packing parts into the box by slowing down the conveyor belt. The key point is that CIRCA is *aware* of its own capacity to deal with a specific combination of goals and environment. This is analogous to the cognizant failure stressed by Gat [13]. Guaranteed control plans also play a crucial role in isolating the unpredictable performance AIS from the rigid, real-time guarantees of the RTS, as discussed above.

Of course, CIRCA's guarantees are based on several assumptions about the generally uncertain, unpredictable real world. However, there is no better way to build a control system: all systems are designed with certain environments in mind, and if they can be proven to manage the specified environments that is only for the better. The uncertainty inherent in the real world makes no difference for this argument. To paraphrase Stankovic [49], the fact that the system may not function correctly or that the world may differ from our environment model with a nonzero probability does not give us license to *increase* the odds of failure by not trying to guarantee performance.

Consider this didactic example: we must transmit vital digital information across a network, and we can use either a simple one-shot transmission or an error-correcting protocol that is guaranteed to correct all known types of errors. Ignoring efficiency (or cost), the error-correcting protocol is clearly the preferable choice, because it has a performance guarantee that the one-shot transmission lacks. This guarantee has value despite the fact that we acknowledge that the protocol is only guaranteed to work for *known* errors. In fact, we can never hope to do better. The task as given is to transmit over a particular network, and the error-correcting protocol has been optimized for that task.

To determine the net value of performance guarantees, we must also examine their two fundamental costs: the one-time cost of making the guarantee and the recurring cost of potentially low utilization. In the case of the error-correcting protocol, these costs might be represented by the time-consuming process of coding the protocol, and the decreased transmission bandwidth available while using the protocol. By both of these measures the error-correcting protocol costs more, but it may be worth the cost to ensure that we really can transmit the information correctly. If the survival of the Space Shuttle depends on the transmitted data, the complex protocol is definitely worth these costs.

One confusing issue is flexibility: is a guaranteed system less flexible than an unguaranteed system? Not necessarily— flexibility and utilization are traded off in guaranteed systems. A complete guaranteed system is maximally flexible because it *must* deal with *all* possible occurrences. This guarantee leads to lower utilization when the environment does not exhibit all of the worst-case behaviors that must be monitored. On the other hand, a system may guarantee to handle only some of the possible occurrences, and in return it could have higher utilization. The flexibility/utilization tradeoff is not unique to guaranteed systems, it is a feature of all bounded-resource systems. The tradeoff is clarified by the fact that guarantees provide a stricter definition of flexibility: a guaranteed system's flexibility can be seen as the fraction of the possible worlds the system is known to be capable of handling. By that definition, an unguaranteed system can only establish flexibility through testing.

In sum, CIRCA's guarantees are only as good as its environment model, and its control plans do incur higher costs than other plans that do not deal with all possible environmental occurrences. On the other hand, CIRCA's control plans have known properties such as correctness and timeliness that can be used in *a priori* analyses, which may in turn lead to modifications in the system's plans and goals. We postulate that, in many complex control tasks, the advantages of guaranteed performance outweigh its costs.

This paper focuses on CIRCA's methods for deriving its safety-guaranteeing control plans. In the following section, we show how the architecture's model of agent/environment interactions forms the basis for these guarantees.

4 The World Model

An efficient world model should represent precisely the information necessary to derive plans, and no more. Since our goal is to derive control plans that are *guaranteed* to meet domain deadlines, these plans must be able to succeed even through the environment's worst-case behavior. Thus the world model we have developed to derive TAP plans is not intended to be a complete, perfect representation of the world's actual behavior; instead, the model represents the world's *worst-case* behavior, and it is used to build plans that can cope with the worst-case. This distinction is extremely important, because it simplifies some aspects of world modeling and motivates the model form we have chosen.

Informally, the world model represents the behavior of the world (including the controlled agent) as movement between *states* via *transitions*. States contain descriptions of the features of the world at some instant, and transitions describe how those features can change. Ongoing processes in the world are represented by "state-encoding"— the status of a process is considered a feature of the world (a "fluent" [28]), and is explicitly encoded into the representation of a state. Important changes in process status thus correspond to transitions between states. Any passage of time that does not lead to significant changes in process status is not represented explicitly: essentially, when no transition occurs the world remains in the same state, where that state may indicate that some process is currently occurring. For example, as the robot arm moves towards the box, the status of this process is encoded into the features `(ROBOT-STATUS MOVING-OVER-BOX)` `(ROBOT-POSITION CHANGING)`. Just continuing to move does not lead to a state change, and thus there is no associated transition. However, when the robot arrives at its destination, the process finishes, the status will change, and the world model will represent this change by a transition to a new state with the features `(ROBOT-STATUS HALTED) (ROBOT-POSITION OVER-BOX)`.

In [34] we briefly introduced a formal representation of the world model as a directed graph. In this paper, we greatly extend the formal representation, showing precisely how control plans can be proven to guarantee the system's safety. The formal world model has five elements (S, F, T_E, T_A, T_T):

1. A finite set of "states" $S = \{S_1, S_2, ..., S_m\}$, where each state S_i represents a description of relevant features of the world.
2. A distinguished failure state F, which subsumes all states that violate domain constraints or control-level goals (e.g., system survival). The system strives to avoid the failure state.
3. A finite set of "event transitions" $T_E = \{T_{E1}, T_{E2}, ..., T_{En}\}$, that represent world occurrences as instantaneous state changes.
4. A finite set of "action transitions" $T_A = \{T_{A1}, T_{A2}, ..., T_{Ap}\}$, that represent actions performed by the RTS.
5. A finite set of "temporal transitions" $T_T = \{T_{T1}, T_{T2}, ..., T_{Tq}\}$, that represent the progression of time. We represent only the significant temporal transitions which lead to state changes.

Each transition $T_i \in T = T_E \cup T_A \cup T_T$ is a mapping between states; $T_i : S \to S$. The functions $D : T \to S$ and $R : T \to S$ determine the domain and range of a transition; $T_i : D(T_i) \to R(T_i)$.

Figure 4 shows an abstracted portion of the graph model for our robot arm domain. Solid single arrows represent event transitions T_{Ei}, dashed single arrows represent action transitions T_{Ai}, and double arrows represent temporal transitions T_{Ti}. State **A** in the figure represents the world state in which the robot has picked up a part from the conveyor and is moving to place it into the box. The double arrow to state **B** represents the continuation of that process until the robot reaches its destination. When the robot arrives over the box, the control system senses that state and halts the motion process, as represented by the dashed arrow to state **C**.

The single solid arrow from state **A** to state **D** represents the possibility that the emergency light may go on while the robot is in motion[4]. From state **D**, the double arrow to state **F** (failure) represents the deadline for reacting to the emergency and pushing the button. The dashed arrows to states **E**, **G**, and **H** represent the planned actions to avoid that failure, quickly halting, placing the part on the table, and moving to push the button. Note that we have modeled these three actions (`HALT`, `PLACE-PART-ON-TABLE`, and `PUSH-EMERGENCY-BUTTON`) as atomic— no event can intervene. Before we can explain why this is necessary, we must first clarify the semantics of state transitions.

4.1 Transitions

At any particular time, the world is considered to occupy a single state in the model, conceptually marked by a unique token w. The token moves instantly along a transition from its

[4] We have omitted other instances of the same event that may occur from state **B** and state **C**.

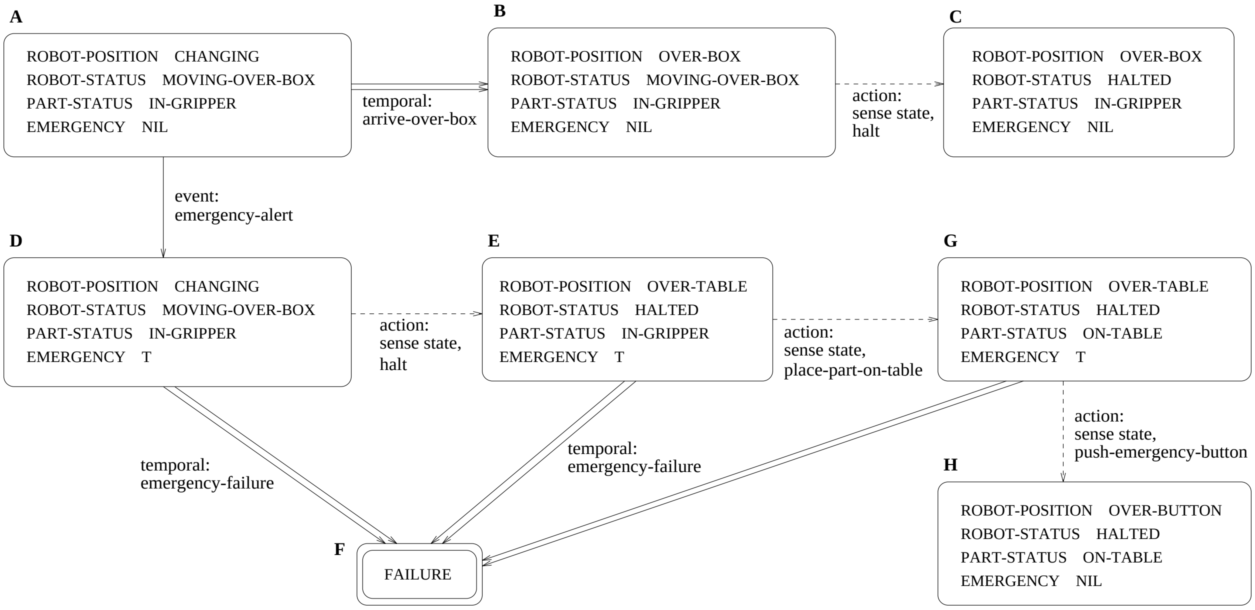

Figure 4: An abstracted portion of the graph model for the robot arm domain. For clarity, many states, state features, and transitions have been omitted.

domain state to its range state when the transition "fires." A transition may fire any time the token is in its domain state and the transition is "enabled." When the token enters a new state, the transitions out of that state are enabled for some interval of time following the transition into that state, as indicated by the function $enabled : T \times \Re \rightarrow \{0, 1\}$. The functions $min\Delta : T \rightarrow \Re$ and $max\Delta : T \rightarrow \Re$ represent the endpoints of the enabled interval as the minimum and maximum delays after the state is entered. So if t_0 is the time at which w enters state S_i, and T_i is a transition leading out of S_i (i.e., $D(T_i) = S_i$), then $enabled(T_i, t) = 1$ for all times t such that $t_0 + min\Delta(T_i) \leq t \leq t_0 + max\Delta(T_i)$.

The different types of transitions have different general forms for their enabled intervals, as shown in Table 2. Since event transitions represent asynchronous and instantaneous external events, which may occur any time the world is in their domain state, their $min\Delta$ is zero and their $max\Delta$ is infinity. Both event transitions and temporal transitions are modeled as uncertain; i.e., they may never fire. This feature prevents the system from building plans that depend on external events or unguaranteed processes for the accomplishment of control-level goals; such dependencies would prohibit any performance guarantees. This is the reason the `PUSH-EMERGENCY-BUTTON` action (among others) must be an atomic transition, rather than a state-encoded process; we must guarantee to push the button to avoid a control-level failure, so the action must itself be guaranteed.

Temporal transitions, by definition, represent the passage of time, and the significant state changes that can occur as processes continue. Temporal transitions have a $min\Delta$ determined by the rate at which the corresponding process is running. In the example of Figure 4, the $min\Delta$ for the temporal transition from state **A** to state **B** depends on how fast the robot is moving, as well as how far it has to move. In this case, the transition's $min\Delta$ represents the earliest possible time

Transition Type	$min\Delta$	$max\Delta$
Event	0	∞
Temporal	> 0	∞
Action	$bcet(\tau)$	$P(\tau) + wcet(\tau)$

Table 2: Enabled interval definitions.

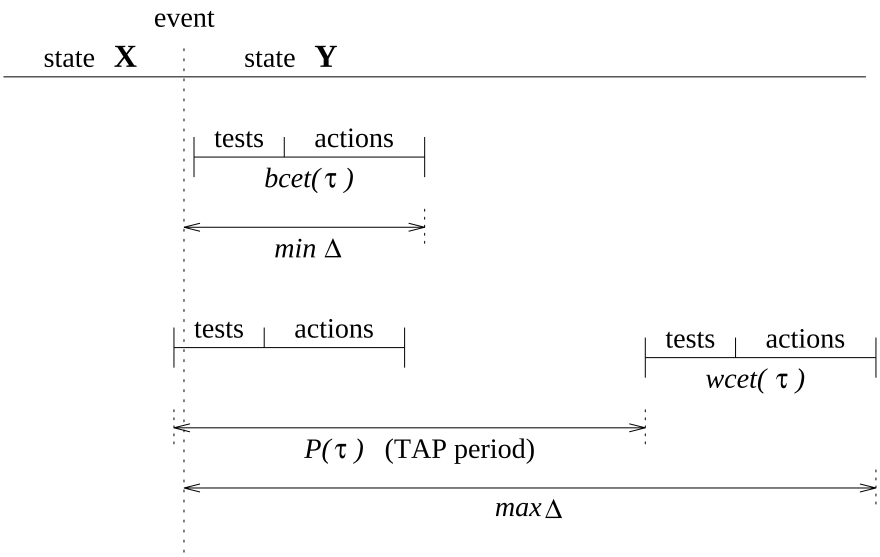

Figure 5: Deriving the $min\Delta$ and $max\Delta$ for an action transition implemented by a periodic TAP τ.

the robot could ever arrive over the box (and thus enter state **B**).

Action transitions represent the intentional activity of the RTS, and thus can have more rigorously defined temporal behavior. In particular, since an action is implemented by a TAP running with a fixed period, we can compute values for the minimum and maximum delay between the time the world enters a state and the time the TAP fires, sensing that state and executing the action. We assume, as a worst case, that a TAP's tests take a "snapshot" of the world when they are first run and spend the rest of $et(tests(\tau))$ processing that captured data. We also assume that the TAP's actions do not actually affect the world until the very end of $et(actions(\tau))$. Thus the minimum delay between entering a state and completing a relevant TAP's actions is $bcet(\tau)$, as illustrated in Figure 5. In the figure, the upper time-line shows the occurrence of an event that moves the world from state **X** to state **Y**. Below that, the $min\Delta$ case is illustrated by a TAP whose tests begin just as the new state is entered. Below that example, another periodic TAP is shown just missing the state transition (its tests started just before the event). In that case, the TAP will not correctly sense the new world state until its next invocation, and thus the action transition implemented by that TAP has a $max\Delta = P(\tau) + wcet(\tau)$, where $P(\tau)$ is the period of the TAP.

4.2 Proving Safety

Given this understanding of the dynamics of the world model, we are now in a position to lend rigor to the notion that some control plans can "cope" with the world. First we will define the goal of a control plan as keeping the world restricted to a particular subset of states, and then we will show how that goal can be provably achieved.

We define an "event-closed" set of states $S_{EC} \subseteq S$ as a set of states for which every event transition from every state in the set leads to a state that is also in the set. That is, $\forall T_{Ei} \in T_E \mid D(T_{Ei}) \notin S_{EC} \vee R(T_{Ei}) \in S_{EC}$. In other words, instantaneous events cannot move the system out of the event-closed set of states; only actions and temporal transitions can leave the event-closed set.

An event-closed set of states with no events leading to the failure state is called a "safe" set of states ($\forall T_{Ei} \in T_E \mid D(T_{Ei}) \notin S_{safe} \vee (R(T_{Ei}) \in S_{safe} \wedge R(T_{Ei}) \neq F)$). Note that a safe set of states can still lead to the failure state through temporal transitions (i.e., it is possible that $\exists T_{Ti} \in T_T \mid D(T_{Ti}) \in S_{safe} \wedge R(T_{Ti}) = F$). These temporal transitions to failure correspond exactly to violating the hard real-time domain constraints: if the system fails to react to a state before a hard deadline, then in the worst case it will enter the failure state via a temporal transition. By "waiting too long" to react, the system fails. In the context of real-time computing, this is known as a *timing failure*.

The definition of a safe set of states is not particularly restrictive, since it only prohibits event transitions to failure and event transitions that lead out of the set. The former requirement is necessary because no system can guarantee to avoid failure if it has no time to react to an event before failure occurs. The latter requirement is intended to allow the system to use action transitions to keep the world within the safe set, never moving to a state from which failure is possible via an event transition. In essence, the existence of a safe set of states only constrains the environment such that an agent must always have some minimum time to react before a failure occurs.

Finally, we can define a "safely-controlled" set of states S_{SC} as a safe set which also has no temporal transitions to failure or out of the set (i.e., $\forall T_{Ti} \in T_T \mid D(T_{Ti}) \notin S_{SC} \vee (R(T_{Ti}) \in S_{SC} \wedge R(T_{Ti}) \neq F)$). The goal of a control plan is to ensure that the world remains in a safely-controlled set of states, so that failure can *never* occur. This is analogous to a stable closed-loop control policy [47] which is known to restrict the operation of a controlled system to a desirable range of states. To show how a control plan can make a safe set of states a safely-controlled set, we now introduce a simple set of correctness-preserving model transformations. These transformations prune out unreachable states [12], and thus allow us to prove safety properties by showing that certain control plans can restrict the world so that no failure states are reachable.

4.3 Model Transformations

We must first define the concept of reachability in our world model. We represent reachability, or the possibility of the world entering a given state, as a predicate $reachable : S \rightarrow \{0, 1\}$, where $reachable(S_i) = 1$ if $\exists T_i \in T, \exists S_j \in S \mid reachable(S_j) \wedge D(T_i) = S_j \wedge R(T_i) = S_i$. This recursive

$\forall S_i \in S - I,\ \ \forall T_i \in T$		
$preempted(T_i)$	$\equiv$	$\exists T_j \in T \mid max\Delta(T_j) < min\Delta(T_i)$
$unreachable(S_i)$	$\equiv$	$R(T_i) \neq S_i$
$unfireable(T_i)$	$\equiv$	$unreachable(D(T_i))$

Table 3: Conditions for removing world model states and transitions.

definition merely says that a state is reachable if there is a transition to that state from another reachable state. We ground the recursion by defining a set of initial world states $I \subset S$ such that $\forall I_i \in I \mid reachable(I_i) = 1$. For any initial state I_i, the transitive closure of reachability from that state yields R_{Ii}, the set of all states reachable from that initial state. In general we do not distinguish among possible initial states, and thus when we speak of the set of reachable world states we mean the union of the reachable sets from each initial state: $R_I = \bigcup_{Ii \in I} R_{Ii}$.

The "correctness" of a world model is determined by how accurately it represents the behavior of the world. In our case, the model is intended to represent all of the worst-case possible behaviors, so the set of all reachable world states R_I is the crucial factor in determining the correctness of our model. If the world model predicts exactly the same states that are possible in the real world, it is most correct. If the model predicts those correct states plus some additional states, the only problem is inefficiency because the system may plan actions to account for states that can actually never occur. However, if the model fails to predict some possible world state, the system may not plan a necessary control action, leading to failure during plan execution. Thus the model transformations we use preserve the model's correctness by never removing model states unless those states can never be reached.

The first, most powerful transformation simply involves removing transitions that are *preempted*; that is, transitions which can never fire because some other transition will always fire first. In terms of our representation, a transition T_i preempts another transition T_j if $max\Delta(T_i) < min\Delta(T_j)$. Since events have $min\Delta = 0$, nothing can preempt an event. Temporal transitions have non-zero $min\Delta$, and thus we can design action transitions (whose $max\Delta$ depends on the frequency we choose for the corresponding TAPs) that will meet the preemption criterion. A preempted transition never becomes enabled and thus can never fire, so it can be removed from the graph model without affecting the correctness of the model.

Two other simple transformations complete the required set. First, it is obvious that any non-initial state that has no transitions leading into it is unreachable, and thus can be removed from the model without affecting correctness. Finally, all transitions leading out of states that are unreachable can also be removed, since they will never fire either. Table 3 summarizes the conditions for these model transformations.

By propagating the preemptive effects of planned control actions into the removal of states from the world model, these transformations show how control plans can force the world to remain within a safely-controlled set of states. Control plans that meet this criterion are called "complete" control plans, and they guarantee that the system will avoid failure.

Beyond this, however, complete control plans also provide one other feature critical to CIRCA's operation. We have previously noted that resource restrictions generally make it impossible to produce a single control plan that will guarantee safety and achieve all task-level goals. Thus CIRCA breaks task-level goals into subgoals and tries to build complete control plans for each subgoal. It is essential that these control plans guarantee to avoid failure and also guarantee to avoid moving out of the safely-controlled set of states for which they were planned, so that the system can continue running a complete control plan for an indeterminate amount of time without risk of violating its control-level goals. Thus the AIS can utilize unpredictable or high-variance AI techniques to build control plans, because while it is building one, the previous control plan is running on the RTS and keeping the system safe.

4.4 Relationship to Petri-Net Models

It is useful to compare this type of state-based model with models based on Petri Nets (PNs) and their variations [38]. In PN models, "places" represent the status of world features, and transitions connect places, representing the way features can change. Multiple tokens can be spread among the places, and the complete state of the modeled world at any instant is defined by the distribution of those tokens, known as the "marking" of the net. Thus in PNs the set of world states that can be reached from any initial world state is represented by the set of net markings that can be reached from an initial marking. In contrast, each state of our world model is a complete description of the world, and the set of world states that can be reached from an initial state is represented by the set of model states reachable from that initial state. In other words, the explicit state enumeration of our world model makes the set of reachable world states extremely easy to recognize.

This feature is desirable because, as we have just shown, reachability is the key to proving safety. In the process of building the world model, it is trivial for the AIS to recognize when a failure state is reachable, because it will actually create a state with the `(FAILURE T)` feature. Thus, *while building* the world model, the AIS can immediately plan actions to avoid failures. Planning for a world model represented as a PN would be considerably more difficult, because the effects of actions on the reachability of particular world states are much harder to determine. In effect, our state-based world model trades the storage space cost of enumerating world states against the computation time cost of determining reachability in a more compact PN model.

4.5 Worst-Case Simplifications: Uncertainty, Determinism, and Time

Because our world model need only represent the worst-case behavior of the environment, several potentially complex representation issues are simplified. For example, a great deal of research has been focused on methods for explicitly representing and propagating uncertainty about the likelihood of various events. Our world model has no need of that information: any possible transitions between world states must be included in the world model, no matter how improbable they are, because in the worst case they just might occur. However, if the system eventually does need to make compromises because it cannot guarantee all of its control-level goals, then having information on the likelihood of various states leading to failure might help the system make intelligent choices about which control-level goals can best be left unguaranteed.

Similarly, uncertainty about the world's initial state is not explicitly represented. Instead, the initial world features specified by the AIS are assumed to match a set of initial model states I, and control plans must be built to deal with all of the states reachable from each of those potential initial states.

As for uncertainty about information from sensors during runtime, the system is required to be able to sufficiently distinguish the current world state whenever an action has been planned. This minimal capability is required by any system claiming guaranteed performance. Note that this does not mean that the precise, complete world state must be determined for action (because some subset of world features may be sufficient to determine the appropriate action— see Section 5.2), nor does it mean that the control system must be able to perfectly track the progression of states in the environment [41]. In fact the system never needs to know the world's state if it does not need to take any action; thus, the world can traverse many transitions but cause no change in the control system. The RTS' internal representation of the world can become quite outdated, but only in non-critical ways.

For example, while the robot arm is responding to an emergency alert, the next part may arrive on the conveyor belt. However, the system may not immediately recognize this event, because it is in the middle of the actions responding to the emergency. These emergency-response actions are scheduled at a higher frequency than the actions that deal with arriving parts. The response latency and the resulting temporarily "out-of-date" internal state of the RTS are non-critical because, even if the system had seen the new part immediately, it would have had to continue the ongoing reactions to avoid a timing failure from the emergency. In the process of building the control plan, the AIS has already examined this sequence of events and has guaranteed that the control plan functions correctly.

We have described the world model transitions with unique range states, but this does not mean that the world or the model must be deterministic. Transitions can also specify multiple possible range states. Again, because all possible states must be handled, nondeterminism does not add complexity: nondeterministic transitions are equivalent to multiple transitions with identical domains and different range states.

As we have seen, the worst-case criterion also removes the need for any detailed representation of time. Complex temporal logics have been developed for reasoning about the relationships between asynchronous external events, simultaneous actions, and the regular passage of "wall clock" time [2, 8, 17, 28, 50]. So far the only timing information we have shown for our world model is the simple worst-case values needed to recognize preempted transitions. There is no need to explicitly represent or reason about the different possible orders of events or actions, because all of those orders are considered equally likely (in the worst case).

Instantaneous events allow our model to represent simultaneity, but they do so by enumerating

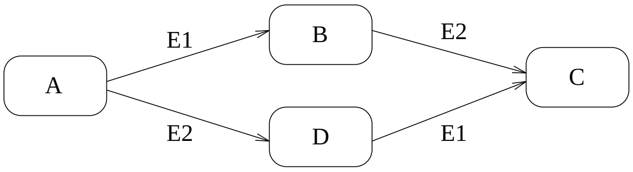

Figure 6: A world model subset showing the representation of potentially simultaneous events.

sequences of states that can occur without the passage of time. For example, in Figure 6 we see that event transitions $E1$ and $E2$ are both applicable to state A. A complex temporal world model might include constraints on the ordering of those events, but that information is of no use to us because the worst case may include any order of occurrence, even simultaneity. Note that the possibility of $E1$ and $E2$ occurring simultaneously is explicitly represented by state C: since the events have $min\Delta = 0$, state C can be entered at the same instant state A is entered.

4.6 Dependent Temporal Transitions

The world model has one difficulty with its minimalist representation of time: dependencies between temporal transitions. To illustrate the problem, Figure 7 repeats a portion of the robot arm domain shown earlier. Beginning with the event `EMERGENCY-ALERT` entering state **D**, the robot has thirty seconds to push the emergency button before failure occurs, as represented by the temporal transition to failure. We can see that taking the necessary actions `HALT` and `PLACE-PART-ON-TABLE` does not remove the threat of failure from the emergency condition. Thus state **E** and state **G** still have temporal transitions to failure. The difficulty is that the minimum time until failure along these transitions is no longer thirty seconds, because the emergency began in state **D**, and some amount of time passed before we halted and moved to state **E**. Thus the real minimum time to failure from state **E** depends on the sojourn time in state **D**. We call this situation a dependent temporal transition, and it complicates the process of reasoning about the world model, as we shall see. However, dependent temporal transitions are still manageable because the worst-case $min\Delta$ for a dependent temporal transition is easy to determine: if T_{Tj} is dependent on T_{Ti} leading out of state S_i, then $min\Delta(T_{Tj}) = min\Delta(T_{Ti}) - max\Delta(T_{Aij})$, where T_{Aij} is the action taken to move between states S_i and S_j. In the figure, the temporal transition from state **E** has $min\Delta = 30$ minus the worst-case execution time of the TAP implementing the action from state **D** to state **E**.

4.7 Action Loops

Mixing action transitions and temporal transitions can lead to one type of pathological subgraph called an action loop. In an action loop, actions join a cycle of states without any intervening events or temporal transitions. For example, Figure 8 shows an action loop that the system might propose while building a plan for the robot arm problem. In the figure, the system has planned to halt in state **D**, transitioning to state **E**. But it has also planned that, once in state **E**, it will immediately resume motion. There are two problems with this action loop. First, the loop can lead to a timing failure because each time the world loops back into state **D**, the time remaining until failure is not the original thirty seconds, but depends on how long it has been since the emergency alert first occurred. CIRCA has no way to recognize when the loop has been executed many times and failure

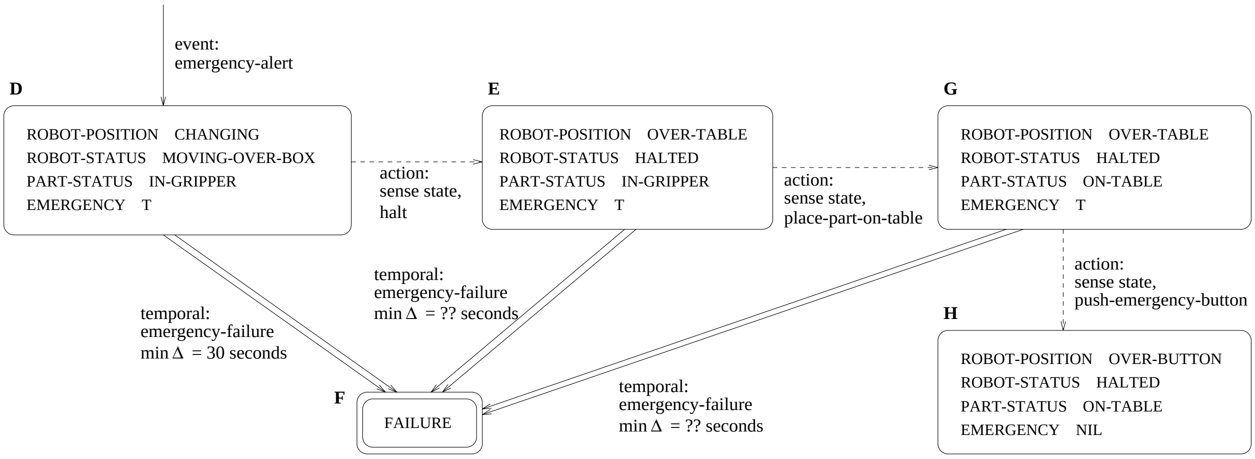

Figure 7: A portion of the example robot arm domain illustrating dependent temporal transitions.

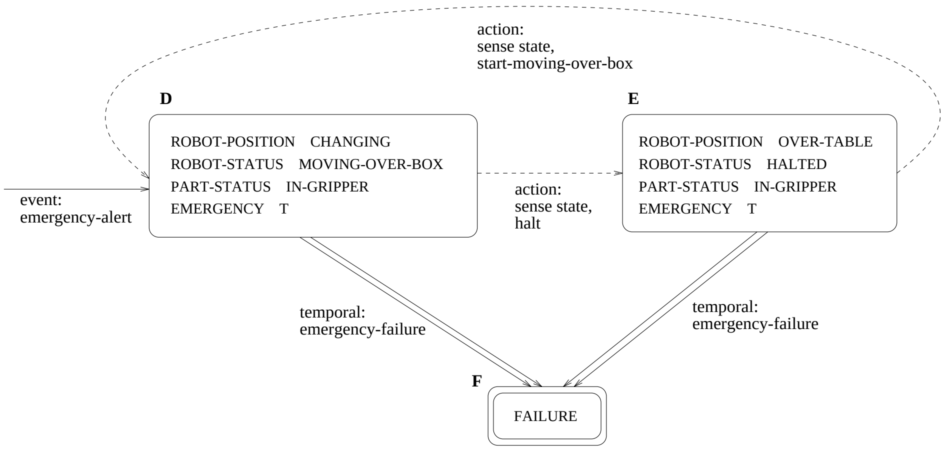

Figure 8: An action loop that might be generated for the robot arm domain.

is imminent.

The second problem with action loops is that they accomplish nothing. In many classical planning systems, an action loop might have a valid purpose because the representation of states is incomplete, and thus side-effects are possible. In our complete state representation, side-effects do not exist, so looping back into a previous state means that the world is *exactly* the way it was (except for the wall-clock time). Thus a sequence of actions leading out of a state and then back into that same state will not accomplish any goal. Note that a loop of states including event or temporal transitions is quite reasonable, because these transitions represent environmental behaviors that may move the world away from desired states, and the system should plan actions to restore those goals.

4.8 Predictive Sufficiency

Figure 9 shows how "inappropriate" TAP actions may be executed if an event occurs between the time a TAP senses the world state and performs its actions. In some cases inappropriate actions

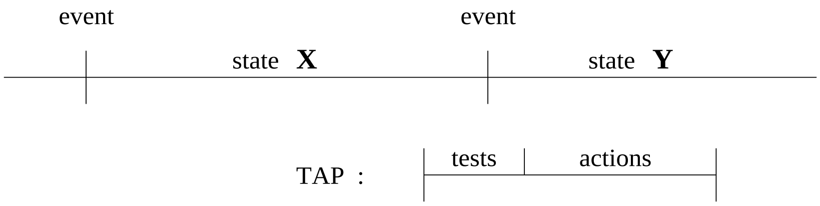

Figure 9: An inappropriate action.

do not matter, and in some cases they can lead to catastrophic failure. Consider an example in which a TAP is used to detonate explosive charges that will demolish a building. Sensors have been installed on the building's doors to make sure that nobody is in the building when it is destroyed. But, as in Figure 9, someone might enter the building just after the sensors are checked, and before the explosives detonate. Since events are instantaneous and asynchronous, the system itself cannot prevent this type of failure. If failure may result from an inappropriate action, we must ensure that the sensors have "predictive sufficiency." That is, a sensor reading must indicate both that a particular condition exists, and that it will continue to exist long enough for the response action to occur ($wcet(\tau)$ in the worst case).

In the demolition example, one solution is to place a ring of sensors several meters from the building, so that people entering the building will first pass through the perimeter sensors. We can then interpret the actual information returned by the sensors ("nobody has crossed the perimeter") to mean that "nobody could enter the building in the next K seconds." The semantics of the sensor data are altered by adding domain knowledge (the perimeter distance and maximum human speed) to yield predictive information, or knowledge about possible future states. We are currently formalizing and implementing techniques by which CIRCA can reason explicitly about the need for predictive sufficiency [35].

4.9 Summary of Agent/Environment Characterization for Guarantees

While we stressed the value of guarantees in Section 3, in this section we have identified critical pieces of information that an agent needs in order to make guarantees about its performance in its environment. The characteristics of the agent/environment interaction that an intelligent, flexible agent must be able to model include:

- Features of the world relevant to the agent, including failure conditions.
- Possible external events, and how they move the world to new states.
- Transitions that are caused by the passage of time, including the minimum time until the transition can occur in the worst case ($min\Delta$).
- All sensing primitives, including their worst-case execution times; sensed data must have predictive sufficiency (as discussed in Section 4.8).
- All action transitions, including their worst-case execution times; actions and sensing primitives must be guaranteed to succeed.

- The set of possible initial states, which must all be safe (or else the agent could fail before it ever begins).
- The actions that preempt temporal transitions, to keep the system in a safely-controlled set of states.

These requirements are not specific to CIRCA's approach to real-time AI; any system seeking to make similar real-time response guarantees must have this information. For example, any system that is attempting to guarantee the timeliness of its behaviors must already have some guarantee that its primitive actions (or some combination of them) will succeed. If primitives are not guaranteed, then it does not matter whether the system decides to act in time, because the action it takes might not affect the environment in the desired way. Similarly, if an agent hopes to avoid failing due to delays, it must be assured that it can take an action between an event and a temporal transition to failure; no system can make safety guarantees in a world with instantaneous transitions to failure.

Guaranteed real-time agent/environment interactions must therefore be characterized as we have described to assure that an agent can achieve its control-level goals in its environment. CIRCA's model captures this characterization and embodies it in a specific architecture. In the context of CIRCA's approach to making guarantees with limited resources, we can also add one more requirement on the agent/environment interactions: it must be possible to partition the state space into safely-controlled sets of states for which the system has sufficient resources. In other words, the RTS is given a control plan that uses limited resources to deal with the contingencies that may arise in a limited set of situations, and we must therefore ensure that the world can be restricted to those handled situations. Intuitively, this means that the domain must afford the opportunity for "stalling" or cycling behavior, where the agent can remain safe by continuing to execute a fixed, limited set of reactions while the AIS is generating the next control plan. For example, a mobile robot can halt and wait for instructions, and remain safe from obstacle collisions with a relatively simple set of reactions. Likewise, in our Puma example, the robot can stack unknown parts on the table, still avoiding failure while it waits for details on how to pack those parts. If this type of task decomposition is not possible, and remaining safe in the environment requires an agent to be continually monitoring for all possible situations, then there is no need for CIRCA's intelligent resource allocation mechanisms.

4.10 Appropriate Domains for CIRCA

Given that CIRCA must model the characteristics listed above, what types of agent/environment combinations will CIRCA be appropriate for? The simple answer is "ones that can be characterized as above," but to get a more intuitive sense, let us reduce the number of criteria. Because CIRCA's world model representation corresponds loosely with the representation of states and transitions in discrete event systems [40], we will employ two terms from that field: *controllability* and *observability*. We will use controllability to reflect the degree to which the intelligent control system can change its current state to another state. Three qualitative levels of controllability are:

fully controllable (the system can take actions to reach any desired state); partially controllable (the system can take actions to avoid undesirable states, but may not have control over all state changes); and uncontrollable (the system has no control over what state it might find itself in). Observability reflects the degree to which the system can determine which state it is in based on its observations/measurements. The qualitative levels here are: fully observable (the system can make adequate observations to uniquely decide which state it is in[5]); and partially observable (the system might only restrict the set of states it believes it might be in).

Finally, because any system has only limited resources, we define a third dimension called *capacity*, which reflects the degree to which a system has enough resource capacity to handle all of the demands placed on it. We will use two qualitative levels of capacity: limited and unlimited.

Along these dimensions, CIRCA is most appropriate for agent/environment interactions characterized as partially controllable, fully observable, and having limited capacity. Examples of such systems include agile manufacturing systems, robotic systems acting in dynamic physical environments (where physical laws impose constraints that allow some controllability), and distributed computing systems. Varying the controllability dimension, CIRCA would also work in fully controllable domains such as control of simple manufacturing robots. However, the complexity of CIRCA could be overkill; a simpler, traditional sequencer should be sufficient to allocate resources as it moves the system deterministically through states. In an uncontrollable system, CIRCA would still be able to guarantee some level of performance given predicted events, but the certainty of these predictions would be limited, and CIRCA's actions might not achieve their goals. Thus, a more reactive approach, like Universal Plans [44]— where more inputs and unguaranteed responses are considered, and outputs are never assumed to be correct— might be more appropriate for such situations. Modeling domains as fully uncontrollable is fairly rare, since that implies that the agent cannot necessarily take actions to preserve its own safety or achieve its goals.

Because CIRCA has been designed specifically to handle limitations in sensing and processing resources by allocating them intelligently, CIRCA would not be an appropriate choice when there are unlimited resources. A single, powerful controller would be more appropriate. In situations where full observability is not assured, making any guarantees on control is problematic. CIRCA could work in such domains by making guarantees based on assumptions about what might be observed, but such guarantees would be inherently probabilistic.

5 Building & Using the World Model: Implementation Details

With this understanding of how control plans can be shown to keep the system safe, we now present CIRCA's implementation of the world model and its methods for developing complete control plans. Our goal here is to describe the unusual features necessary to build real-time control plans using our model of agent/environment interactions. We describe algorithms that successfully implement these features, but we do not contend that these are the most efficient or novel mech-

[5] Note that this implies both the ability for sensing to be sufficiently discriminating and for observations to be sufficiently predictive.

anisms possible. Note that this section describes only the low-level control planning done in the AIS; more complex high-level reasoning is done by a different process. The prototype high-level reasoning mechanism, described fully in [31], is based largely on PRS [14, 20].

From the description of the world model above, we might derive a simple approach in which the entire world model state space is enumerated and then actions are planned to reduce the graph to a safely-controlled subset. Of course, the immediate objection to this approach is that it involves generating and storing a complete enumeration of the state space, which is exponential in the number of world features. Furthermore, since planning a single action can make large sections of the world model's entire graph unreachable, much of that enumeration might be wasted.

Therefore, we have developed an algorithm that dynamically interleaves the construction of the world model and the planning of control actions. The control plans (TAP schedules) that are run on the RTS are developed by five processing phases, outlined below and described in more detail in the following sections.

In the first phase (**planning actions**), the AIS builds up a list of actions that must be taken to make all failure states in the world model unreachable. This phase actually builds and manipulates the world model states on-the-fly, as it is planning actions and simulating transitions. Associated with each planned action is a list of the states to which that action must be applied and the associated temporal transitions which it has been planned to preempt.

In the second phase of processing (**minimizing tests**), the AIS attempts to maximally generalize the preconditions for each action, so that as few tests as possible are necessary to decide when to apply the action. The third phase (**planning sensing**) builds TAPs that perform the tests using selected sensing actions. The fourth phase (**assigning TAP periods**) chooses TAP periods so that they will always preempt their associated temporal transitions, and the final phase (**scheduling TAPs**) invokes the Scheduler to build a cyclic TAP schedule that meets all of the TAP timing requirements.

These processing phases do not operate in a purely feed-forward manner; rather, control and information can flow back from later phases when problems are detected in the developing TAPs. For example, when phase three runs to plan sensing actions it may find that the sensing actions required to test a particular planned action's preconditions are so complex and time-consuming that the action can never preempt the temporal transition it was designed for (i.e., $wcet(tests(\tau)) + wcet(actions(\tau)) > min\Delta(T_t)$). This condition was not detected earlier because the sensing actions cannot be planned until the second phase has minimized the set of feature tests required. Since the temporal transition is no longer preempted, the world model is no longer safe, and the system must backtrack to choose different sensing actions or even different actions altogether.

To allow control (backtracking) to propagate between these different processing phases, we have implemented them in an explicit-stack state machine, illustrated in Figure 10. The action planning and postprocessing phases are cast in the form of individual functions for each decision process—every decision made by the system maps to a function call. The main loop of the system chooses which decision function to run next based on a global mode variable. Each decision function

```
(defun run-control-planner (&aux result)
  (do-until (equal *mode* 'end)
      (setf result
          (case *mode*
                (plan-action (plan-action))
                (check-intermediate-plan (check-intermediate-plan))
                (generalize-tests (generalize-tests))
                (assign-sensors (assign-sensors))
                (build-taps (build-taps))
                (schedule-taps (schedule-taps))))
      (if (null result) (backtrack-all))))
```

Figure 10: The main loop for the AIS control-level planner.

computes its decision, pushes the alternative choices for that decision onto a *choice-stack*, sets the mode variable to select the next decision that should be run, and returns a boolean indicating whether backtracking should be initiated. For example, the basic action-planning decision function looks at the world model state currently being examined, chooses an action to apply to that state, pushes the alternative actions onto the choice-stack, and returns `true`. Or, if there are no more action alternatives for the current state, the function returns `nil`, indicating that backtracking is required. Backtracking affects the world model, the choice stack, and the stack that maintains the state of the decision loop (including the current mode and the current world model state).

By casting the main processing loop in this form, we have made the system highly modular, so that additional decision processes (like postprocessing phases) can be added easily. The explicit stack of this implementation also has an advantage over recursive implementations, because in this formulation it is fairly easy to interrupt and resume the control-level planner.

5.1 Planning Actions

Because the world model state space is exponential in the number of world features, the AIS mechanisms that build TAP plans are actually given a much more compact representation of the world. The input to these mechanisms is divided into three types of information: transition descriptions, initial state descriptions, and goal descriptions. Transition descriptions are simple production rules that detail the changes the world can undergo, much like STRIPS operators [36]. Figure 11 shows example rules from the robot arm domain. Note that the preconditions and postconditions need not fully specify all features of the states to which the transitions apply. These descriptions are implicitly generalized by the lack of certain feature specifications. Action transition descriptions also include information about their worst-case execution times and the required actuator resources.

Goal descriptions also do not usually specify the entire state of the desired world: in fact, many describe just a single feature (such as `(PART-STATUS IN-BOX)`). These partial descriptions are not expanded into an explicit set of acceptable states; instead, the AIS uses the descriptions as litmus

```
EVENT emergency-alert
      PRECONDS: ((emergency nil))
      POSTCONDS: ((emergency T))

TEMPORAL emergency-failure
      PRECONDS: ((emergency T))
      POSTCONDS: ((failure T))
      MIN-DELAY: 30 [seconds]

ACTION push-emergency-button
      PRECONDS: ((robot-status free) (gripper-status free))
      POSTCONDS: ((emergency nil))
      RESOURCES: (arm)
      WCET: 3.5 [second]
```

Figure 11: Example transition descriptions given to the AIS.

tests for states which it generates on-the-fly, as detailed below.

5.1.1 The Planning Algorithm

Given this input information, the AIS dynamically constructs the graph model and the plan of actions together in a single depth-first search process, essentially similar to a forward-chaining STRIPS planner [36]. This process operates on a stack of states (the *state-stack*), examining each state in turn and planning actions that achieve goals and preempt temporal transitions that lead to failure. To initiate the processing, each of the completely specified initial states is pushed onto the state-stack. Then, as long as the stack is not empty, the AIS pops a state off the stack and considers it the *current state*. If the current state is unreachable[6], the AIS will ignore it and pop the next state off the stack. If the current state is reachable, the AIS finds all the event transitions and temporal transitions that apply to the current state. The applicable transitions are simulated by substituting their postconditions into the current state description, yielding either new states that have not been examined yet or old states that have already been processed (i.e., states for which actions have already been planned). New states are pushed onto the state stack, while old states are simply updated with the information that they have a new source state.

The AIS then finds all the *acceptable* action transitions that could be taken from the current state. If there are no temporal transitions to failure from the current state, then all action transitions that apply to the current state are acceptable, including the null action `NO-OP`. If there are any temporal transitions to failure, only action transitions that can be implemented quickly enough to preempt the failure are considered acceptable. The AIS chooses from amongst the acceptable

[6] A non-initial state on the stack may become unreachable if actions are planned to preempt every temporal transition leading into that state, and no event or planned action transitions lead into that state.

actions the one that leads to the best next state, as determined by a heuristic scoring function (described below). The other acceptable actions are retained on the choice-stack, so that the next-best alternative will be chosen if the system later backtracks to this point in the search.

Chronological backtracking is initiated when one of two conditions is satisfied. First, the system backtracks when it detects an action loop (see Section 4.7). Whenever a planned action leads to a state S_{old} that has already been processed, the system searches for action loops by looking back recursively along the action transitions leading to the current state, checking to see if any originated at S_{old}. The second condition for backtracking is the recognition that there are no remaining action choices for the current state. In that case, it is clear that the planner has found an unavoidable failure: either there are no acceptable actions to preempt a temporal transition to failure from the current state, or the system must have explored all possible worlds beyond this state and backtracked to reach this state, otherwise `NO-OP` would be a choice.

By running the planning process until the state-stack is empty, the AIS simulates out all of the paths the world might feasibly take while the agent is controlled by a particular set of action transitions. More importantly, that set of action transitions is dynamically defined as the AIS works, in response to the recognition that a failure state is reachable. The basic action-planning algorithm terminates when no failure state is reachable. Using chronological backtracking to consider every applicable action at each state, the AIS can perform a complete search of the set of action plans.

5.1.2 Complexity

We noted earlier that the complexity of some environments may make it impractical to enumerate all possible situations. This is one of the arguments frequently used against *ad hoc* real-time systems that are simply tested exhaustively to demonstrate that they meet hard deadlines [49]. How, then, does CIRCA's enumerative world modeling technique differ?

The most important difference is that the AIS *does not* enumerate the entire domain state space. As discussed earlier, the AIS' high-level planning explicitly divides long-term goals into shorter-term subgoals, which are then separately implemented by control plans. This restricted context means that the state space of the control planner is not the entire set of states the system and world can ever enter.

Furthermore, the planner avoids enumerating even this restricted space because, while it is generating the world model, it is also generating the plan of actions. Each time an action is planned, it restricts the world's behavior and thus prunes out states that the AIS never even considers. In our Puma domain, one of the problem variations has a complete model space of over 5100 world states. To build a complete control plan that guarantees all control-level goals and also achieves the task-level goals, the AIS only enumerates 330 unique states. The final plan restricts the world to a safely-controlled set of 158 possible (reachable) states. For a problem in which the world is described by eleven different features, and eight actions are planned for 144 different states, the size of the space actually searched seems quite reasonable.

Even if the system searches as much as possible and cannot produce a complete control plan, it is highly unlikely that the entire search space will ever be enumerated. The reason is simply that CIRCA only enumerates *possible* world states. In most realistic worlds, the structure of the external world makes many combinations of world features impossible. This is reflected in the AIS' world model by the fact that many combinations of state features are not reachable, even without any planned actions. In our example domain, it is not possible for the features `(OBJECT-STATUS IN-GRIPPER)` and `(GRIPPER-STATUS FREE)` to coexist. This fact is not explicitly represented, and the AIS never generates a state with those features and then realizes it violates a domain rule; instead, the system simply cannot generate that state because it is not possible with the given transition descriptions.

In general, any system making guarantees must somehow ensure that those guarantees hold for all possible worlds. This requires either an exponential enumeration of states or some dependency information that allows the system to extend guarantees made for one state to other states without examining the others individually. Recent work by Godefroid and Kabanza [15] illustrates one way in which such dependency information can reduce search spaces; their results allow a system to examine only a single ordering of independent actions, rather than enumerating all possible orderings. These results are not immediately applicable to CIRCA, because their world model does not include external events. This omission simplifies the concept of action independence to a condition on the action descriptions. In the CIRCA model, this condition alone is not sufficient to determine if actions are independent: by enabling or disabling event transitions, an action can affect another even if its description includes no overlapping terms. We are actively investigating ways of deriving independence conditions in CIRCA's model of agent/environment interactions.

However, the most important point to remember is that the planning done by CIRCA's AIS is isolated from the real-time domain deadlines. The AIS *does not* need to meet deadlines while producing control plans, so the complexity of the planner is decoupled from the agent's interactions with the world. In fact, the complexity of planning is one of the fundamental motivations for CIRCA's distinction between the AIS and RTS: the high-variance search for plans to achieve goals must be isolated from ongoing, real-time interactions with the environment.

5.1.3 Incremental Improvement

Currently, the system makes only a crude distinction between control-level and task-level goals. All control-level goals must be achieved, or the system backtracks. If some task-level goals are not achieved by a control plan, the system may still consider the plan acceptable. In the future, we may add more information so that the system can make intelligent decisions about risk-taking in the pursuit of task-level goals. This information might include criticality ratings for goals and event probabilities, so that the system could compute the utility of guaranteeing different subsets of control-level goals. In general, however, our initial focus on guaranteed behavior has led us to ignore such difficult information; we have concentrated instead on developing a system that can make rigid, complete guarantees within the scope of its limited knowledge. Given that most rigorous capability, we can easily modify the system so that it can forgo various goals when necessitated by resource restrictions [34].

```
(defun check-intermediate-plan ()
  (let ((plan (find-all-planned-actions))
        (states (remove-if-not #'state-is-reachable-p (find-all-states)))
        (goals-done 0))

  (dolist (goal *goals*)             ;;; Count goals that are reachable.
          (if (any #'state-has-feature-p states goal) (++ goals-done)))

        ;;; If current plan did better than stored, or have none stored yet,
        ;;; store this one.  Stored in global as a list (plan goals-done).
  (if (or (not *stored-plan*) (> goals-done (second *stored-plan*)))
      (setf *stored-plan* (list plan goals-done)))

  (cond ((= goals-done (length *goals*))      ;;; If all goals reachable,
         (setf *mode* 'generalize-tests)      ;;;   move on to next phase
         T)                                   ;;;   and don't backtrack.
        (T nil))))                            ;;; Else, backtrack for new plan.
```

Figure 12: The `check-intermediate-plan` decision function, implementing an incremental improvement method.

With the action-planning algorithm described above, we can derive every possible action plan that guarantees to avoid control-level failure. What we really want, if possible, is a plan that guarantees the control-level goals and also either guarantees or at least makes possible the task-level goals. To find those plans, we have formed the action-planning algorithm as an imprecise computation [24, 32] that will continue generating new plans until no more are available, or until a plan that achieves all of the task-level goals is found. In the current implementation, a plan is considered to achieve a task-level goal if any state satisfying that goal is reachable. The decision function `check-intermediate-plan`, illustrated in Figure 12, is placed in the loop shown in Figure 10, to be run after the `plan-action` phase runs out of states to plan for. If the current plan does not achieve all of the control-level goals and make the task-level goals at least reachable, the decision function returns `nil` and the system backtracks to find a better plan. A more restrictive criterion might test to make sure that task-level goals are reachable from all states in the world model, or that the control plan always drives the system towards the task-level goals.

If the AIS decides, based on task-level time pressures, that it needs to produce the next control plan quickly, it can interrupt the planning loop of Figure 10 and use the current acceptable plan stored in `*stored-plan*`. If the AIS has more time available, it can continue producing plans for as much time as is convenient, and then use the best plan stored so far. In this way, the AIS can itself implement an anytime planning algorithm [7, 43]. This feature is useful because, although achieving control-level goals is never dependent on timely responses from the AIS, achieving non-critical, task-level goals may be. For example, in our box-packing scenario, the system implements the control-level goal of making sure that nothing falls off the conveyor belt by (in the worst case) putting the part it is currently holding down on the table. The control plan must also be able to stop the conveyor when the table is full. When that happens, the robot will continue to satisfy its control-level goals (even easier with the conveyor stopped!), and no catastrophes will occur. However, the faster the AIS figures out how to pack the articles sitting on the table, the faster the system will achieve its task-level goal of generating a packed box.

5.1.4 The Scoring Heuristic

The scoring function used to choose actions is the only heuristic knowledge currently used by the control-level action planner. The heuristic performs a recursive N-step lookahead, returning a value corresponding to the best state reachable in N transitions from the current state. The scoring function expresses preferences for states based on how completely they satisfy the system's control-level and task-level goals. Since control-level goals are defined to be those which the system is trying to guarantee, they are weighted as more important than task-level goals. In fact, we consider violations of control-level goals to be equivalent to risking the safety of the system, and thus a violation of any single control-level goal is considered worse even than a violation of all the system's task-level goals.

The planner may choose an action that leads into a state from which a temporal transition leads to failure. Clearly, the longer the $min\Delta$ of that temporal transition to failure, the easier it will be to avoid failure by taking another action. Thus the scoring function also expresses a preference for states which have the longest possible delays until failure occurs. To guide the system towards choosing the shortest path to success, the scoring function also takes into account the number of transitions which must be traversed to reach a state with a desirable set of features.

5.2 Minimizing Tests

Because an action may be useful in several world states, we do not build up complete TAPs with sensing requirements as soon as an action is planned: if the action applies to several states, we would end up with multiple TAPs implementing the same action with different, but probably similar tests. This would make the scheduling operation much harder. Instead, we wait until all of the actions have been planned, and we have a full description of their sets of domain states. Then, in the second phase of processing, the AIS attempts to maximally generalize the preconditions for each action, so that as few tests as possible are necessary to decide when to apply the action. This phase is especially crucial when actions are applied to several states: the minimization phase can eliminate the need to test some specified features if the omission of those tests will not allow the action to be applied to a state for which it was not planned.

The test minimization process is essentially equivalent to the minimization of switching circuits [22]. Each action can be considered separately as a circuit whose minterms are the features of the states for which it has been planned. All states that are not reachable in the world model are considered "don't-cares," because it does not matter whether the final testing expression includes their features or not, they can never occur.

```
SENSOR overhead-camera
   DETECTS: (part-seen part-type robot-position)
   WCET: .1 [seconds]

V-SENSOR robot-status?
   DETECTS: (robot-status)
   P-WCET: .02 [seconds]
   USES: ((overhead-camera 1)(moving? 1))
```

Figure 13: Example sensor and virtual sensor descriptions.

For example, in the robot arm domain, the planner initially plans to take the action `PUSH-EMERGENCY-BUTTON` in four states, each of which has eleven features. After minimization, the action is associated only with tests for `((EMERGENCY T) (GRIPPER-STATUS FREE))`. The new tests do not check all eleven state features, so they will take less time to execute. Of course, with only those two preconditions, the resulting TAP will match many more than the originally planned four world states. However, the minimization algorithm has determined that none of those additional matching states are reachable, and thus they do not matter. Note that the minimization phase can even remove preconditions that are *required* to execute the action. In this example, the `PUSH-EMERGENCY-BUTTON` action transition description in Figure 11 included the precondition `(ROBOT-STATUS FREE)`, but that precondition was removed during minimization because it is not needed to distinguish the four planned states.

The general test minimization problem is NP-complete, so we have avoided using a complete algorithm. Instead, the minimization phase is implemented using the heuristic ID3 program[7] [39], which is given the states for which an action has been planned as positive examples and all the other planned (possible) states as negative examples. ID3 incrementally builds a decision tree to distinguish the positive examples from the negative examples. While this approach does not guarantee an optimally small decision tree, it yields reasonable results with very little processing.

5.3 Planning Sensing

Once the action preconditions have been minimized, the AIS plans sensing actions to implement the precondition tests. To plan sensing actions, the AIS examines descriptions of the system's sensors that include what world features the sensor detects and its worst-case execution time. Figure 13 shows two example sensor descriptions.

The first example describes a physical sensor in the system, the overhead camera that returns information about arriving parts and the position of the robot. The second example describes a "virtual sensor," a software construct that may access several physical sensors (and/or several readings from a single sensor) and combine their values. In the example, the virtual sensor `robot-status?`

[7] Marcel Schoppers suggested this approach.

combines single readings from the camera and another virtual sensor (`moving?`) to determine the robot's status. The worst-case execution time for the virtual sensor is determined by adding the time needed to access the component sensor values to the worst-case processing time, indicated by `P-WCET`.

Virtual sensors can also access the limited RTS world model, which is essentially a set of storage locations that hold status information. For example, the virtual sensor `moving?` accesses an RTS storage location to determine whether the robot is currently moving. The actions that start and stop motion also set the value of this storage location. No physical sensor readings are required, and thus the `moving?` virtual sensor executes very quickly.

One of the areas in which CIRCA is currently being extended is the automatic assignment of additional internal storage locations to buffer physical sensor readings that will be useful to future precondition tests. If a physical sensor reading is fairly costly to acquire and its value is known to persist for a sufficient time, then several actions that test that value in their preconditions could instead access the stored result of a single physical sensor execution. This automatic planning of the use of internal storage to avoid excessive sensing could greatly enhance the system's efficiency, allowing the AIS to produce TAP schedules for domains which would otherwise be too demanding.

Some systems may have multiple sensors capable of detecting a particular world feature, and some sensors may detect multiple world features. Thus the task of assigning sensors to action preconditions is a covering problem, involving finding a minimal set of sensing actions that will test all the preconditions. The AIS could solve this problem via a depth-first search process over all the possible covering sets. Each covering set would be checked to make sure that, when combined into a TAP, the resulting worst-case execution time does not exceed the $min\Delta$ of the temporal transition the action has been planned to preempt. If it does, the system would backtrack to try the next possible covering set of sensing actions. If no set of sensing actions could be built to yield a sufficiently short TAP, then the backtracking would propagate back to the previous processing phases, and the system would search for a different control plan.

Currently, the sensor-planning functionality has been implemented in the Puma domain in a simplified form. Rather than performing a search, an association list is used to map abstract world model state features to different combinations of sensed real-world features. The higher-level AIS processing can modify this association list as necessary, bypassing the search processing that might otherwise be used to perform sensor planning.

5.4 Assigning TAP Periods

Once the sensing actions have been chosen, the complete set of TAPs is built and their worst-case execution times are available. In the final phases of processing, the AIS assigns periods to the TAPs and builds schedules that meet those periodic constraints. Assigning TAP periods is largely a trivial task, except for TAPs that deal with dependent temporal transitions. For other TAPs, the preemption equation described earlier shows that each TAP's period should be just less than the corresponding temporal transition's $min\Delta$ minus the TAP's worst-case execution time.

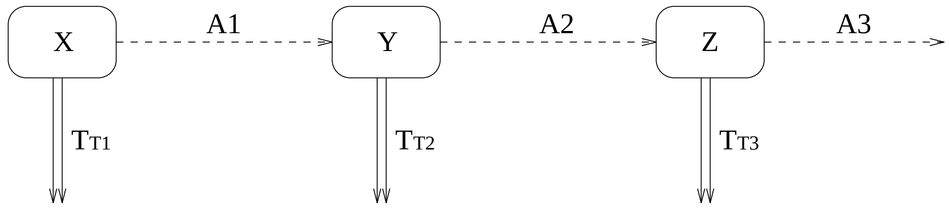

Figure 14: Example actions dealing with dependent temporal transitions.

For TAPs dealing with dependent temporal transitions, the problem is complicated by the dependencies between TAP periods. For example, Figure 14 shows a chain of temporal transitions where T_{T1} is the initial temporal transition applicable to state **X**, and the actions $A1$ and $A2$ do not remove the cause of the temporal transition. Thus dependent versions of the temporal transition apply to the succeeding states **Y** and **Z**. As presented earlier, it is easy to compute the minimum delays until the dependent transitions are enabled:

$$min\Delta(T_{T2}) = min\Delta(T_{T1}) - P(\tau_{A1}) - wcet(\tau_{A1})$$

$$\begin{aligned} min\Delta(T_{T3}) &= min\Delta(T_{T2}) - P(\tau_{A2}) - wcet(\tau_{A2}) \\ &= min\Delta(T_{T1}) - P(\tau_{A1}) - wcet(\tau_{A1}) - P(\tau_{A2}) - wcet(\tau_{A2}) \end{aligned}$$

where τ_{A1} and τ_{A2} are the TAPs that implement the respective actions. In the general case, where n actions are needed to end the chain of dependent temporal transitions, we see that

$$min\Delta(T_{Tn}) = min\Delta(T_{T1}) - \sum_{i=1}^{n-1} [P(\tau_{Ai}) + wcet(\tau_{Ai})]$$

We also know that, for the preemption condition to hold for the final action An that terminates the chain, we must have $min\Delta(T_{Tn}) > P(\tau_{An}) + wcet(\tau_{An})$.

Substituting, we see that

$$min\Delta(T_{T1}) > \sum_{i=1}^{n} [P(\tau_{Ai}) + wcet(\tau_{Ai})]$$

This equation essentially shows that the $min\Delta$ of the initial temporal transition must be long enough to accommodate all TAPs invoked in the dependent chain. Rearranging the equation to solve for the periods, we have

$$\sum_{i=1}^{n} P(\tau_{Ai}) < min\Delta(T_{T1}) - \sum_{i=1}^{n} wcet(\tau_{Ai})$$

In other words, the sum of the TAP periods must be less than the total *slack time* remaining in the original temporal transition when all of the TAPs use their worst-case execution time. Unfortunately, we cannot solve this equation alone for the TAP periods because there are n free variables and only one independent equation. Thus additional constraint equations must be added. We synthesize those constraints based on the observation that scheduling periodic tasks is easier if their utilization is low; that is, if their execution times are relatively small compared to their periods. To keep each TAP's utilization low, the choice of each TAP's period should be influenced by the length of the TAP's execution. For example, assigning a short period to a complex, costly TAP will leave little slack time between its invocations for the other TAPs to run. Thus longer TAPs should be given longer periods, and shorter TAPs can be given shorter periods without leading to excessively high utilization. To achieve this effect, we distribute the total slack time among the TAP periods in proportion to each TAP's worst-case execution time:

$$P(\tau_{Ai}) < \frac{wcet(\tau_{Ai})}{\sum_{j=1}^{n} wcet(\tau_{Aj})} \left[min\Delta(T_{T1}) - \sum_{j=1}^{n} wcet(\tau_{Aj}) \right]$$

So, for chains of states with dependent temporal transitions, the system adds up the total worst-case execution time for the TAPs in the chain, subtracts that from the $min\Delta$ of the first temporal transition in the chain, and divides the remaining slack time proportionally among all of the TAPs. This distribution has the effect of making each TAP have the same utilization.

Unfortunately, the intuitive motivation for this equal-utilization strategy is not entirely accurate: it is not always best to have TAPs with equal utilizations, particularly when TAPs may have widely-varying worst-case execution times. For example, consider two TAPs, A and B, with worst-case execution times of 10 and 100 milliseconds respectively. Suppose that these two TAPs are required to preempt a dependent temporal transition chain with $min\Delta(T_{T1}) = 500$ milliseconds, as described above. Using the equal-utilization strategy, TAP A would be assigned a period of $(10/110) * (500 - 110) \approx 35$ msec[8]. However, it is immediately obvious that this will not lead to a feasible schedule, because $wcet(B) > P(A)$. No schedule will ever be possible if this condition holds, because any invocation of TAP B would immediately imply that TAP A had missed its deadline.

Therefore, it is clear that every TAP must have a period that is at least greater than the maximum worst-case TAP execution time ($wcet(T_M)$) that will be scheduled. We can incorporate that requirement into our period assignment strategy by pre-allocating at least that much time to each TAP period:

$$P(\tau_{Ai}) < wcet(T_M) + \frac{wcet(\tau_{Ai})}{\sum_{j=1}^{n} wcet(\tau_{Aj})} \left[min\Delta(T_{T1}) - \sum_{j=1}^{n} wcet(\tau_{Aj}) - n * wcet(T_M) \right]$$

[8] Note that we truncate the actual computed value to maintain the required inequality.

For the example TAPs, this results in setting $P(A) = 100 + (10/110) * (500 - 110 - 2 * 100) \approx 117$ and $P(B) = 272$. These period assignments lead easily to the simple feasible schedule AB.

While this simple two-TAP example works well, experiments have shown that, when more TAPs are being scheduled, the TAP periods may still be assigned so that shorter TAPs have periods that are too short to allow enough other TAPs to execute between invocations. Thus it has proven useful to increase the pre-allocation of time to all TAPs above and beyond the required $wcet(T_M)$. The amount of this increased allocation is determined by multiplying $wcet(T_M)$ by a value greater than one. For the Puma domain, a multiplicative factor of 1.2 has provided the best performance, although experimentation was limited to a few scheduling problems.

While this approach to assigning TAP periods is designed to make scheduling the TAPs as easy as possible, other considerations might usefully influence the period-assignment phase. For instance, if the various states in the chain have different levels of desirability, it might be preferable to bias the TAP periods so that the system spends more time in the preferred states. In the example of Figure 14, if an event led from state **Y** to a highly-valued new state, it might make sense to increase the period of τ_{A2}, so that the system might remain in state **Y** longer, giving more time for the beneficial event to occur.

5.5 Scheduling TAPs

In the final phase of generating TAP control plans, the AIS sends the accumulated information about the TAPs to the Scheduler module. The Scheduler tries to build a cyclic schedule that runs TAPs at least as frequently as their periods require. In the current implementation, the RTS can run only one TAP at a time, and TAPs are not interruptible, so the Scheduler does not need to consider TAP preemption. The Scheduler uses a modified deadline-driven scheduling algorithm [25] to optimally derive a TAP schedule. The basic deadline-driven algorithm specifies that, each time the system can choose which TAP to run, it should run the available TAP with the closest deadline. To derive a cyclic schedule with this criterion, the Scheduler simulates the operation of a dynamic scheduler, incrementing a time counter and deciding which TAPs to run as simulated time passes. After the simulation has progressed far enough to invoke the TAP with the maximum MAX-PERIOD, the Scheduler begins scanning the trace of the simulation, attempting to extract a loop of TAP invocations which meets all TAP timing requirements. The maximum possible loop size is equal to the least common multiple of the TAP MAX-PERIODs. If the Scheduler cannot build a successful schedule to guarantee all the TAP timing constraints, the AIS backtracks to generate a different proposed TAP plan.

6 Related Work

In this section, we discuss several research projects that relate closely to CIRCA's world-modeling methods and the way CIRCA generates performance guarantees. A more general comparison to related work on combining AI and real-time control appears in [34], and a survey of the entire field appears in [23].

While a number of recent research projects have focused on interactions with the real world, and even with combining traditional strategic planners with reactive systems, relatively few of these projects have made any mention of the real-time nature of their environments. Most reactive systems simply execute as fast as they can, and they are engineered to perform "fast enough" for a given environment. As we argued in the Introduction, a general architecture for intelligent real-world interactions should provide greater flexibility by *proving* that a particular system has sufficient capacity to meet its environment's deadlines.

CIRCA's mechanisms for providing these proofs are the result of combining two fundamental techniques. First, the RTS provides a predictable basis for real-time guarantees by running low-variance reactions in a fixed schedule. Second, the AIS generates those schedules of reactions and proves that they will maintain the system's control-level goals.

6.1 Predictable Execution

In this section we discuss the rare AI systems that have been interfaced to the real world and have dealt with the concepts of hard deadlines and predictable execution. These systems variously rely on real-time operating systems, constant-cycle-time circuits, or any-time algorithms to enforce guaranteed, predictable execution.

6.1.1 DR/MARUTI

Hendler and Agrawala [19] are integrating an enhanced Dynamic Reaction (DR) system and the MARUTI operating system to implement guaranteed real-time reactive reasoning in a manner very similar to CIRCA's guaranteed TAP schedules. The DR system sets up asynchronous monitor processes to check conditions on specific world model features: signals from these monitors drive changes in reactive activities. The MARUTI operating system provides explicit support for scheduling hard real-time tasks on distributed systems, guaranteeing the execution of jobs that are accepted. By using MARUTI to schedule and execute the reactive elements of DR, the combined system can make performance guarantees similar to those CIRCA provides for its control-level goals.

Higher levels of planning have been added to the DR model using the notion of abstraction: the reactive system reasons about detailed information in very small units of time, while higher levels of reasoning use more abstract data and larger time scales [18]. Complex reasoning is implemented by reactive elements that are triggered by abstract information in the world model. The enhanced DR model thus attempts to smoothly integrate reactive reasoning and higher-level reasoning within a single processing model, unlike the abrupt distinction CIRCA makes between task-level and control-level goals. While this integration is desirable, it blurs the notion of guaranteed execution, because it is not clear which reactive elements must be guaranteed and which not. By separating the AIS and RTS, CIRCA avoids this issue but must carefully limit the communication between the subsystems to avoid jeopardizing its performance guarantees.

DR/MARUTI currently does not reason about its scheduling requirements: it does not generate them, and it cannot revise them if sufficient resources are not available. However, Hendler and Agrawala have expressed interest in methods for internally deriving the scheduling requirements of the system [19], much as CIRCA reasons about TAP requirements. They discuss the need to increase the flexibility of DR/MARUTI so that it may include non-real-time jobs, just as CIRCA provides the unguaranteed TAP list. They also note that a "context-switching" approach might be used to switch between predetermined reactive schedules based on environmental data. This is precisely the way in which CIRCA operates continuously: it builds TAP schedules off-line from the execution unit (in the concurrent AIS) and the RTS executes each schedule when the environment has reached the appropriate point in the plan.

6.1.2 CROPS5

CROPS5 is a C-based parallel implementation of the OPS5 production system [37]. The production system is encapsulated within an "AI server" program that runs under a real-time operating system, allowing the production system to run only when other, guaranteed real-time control tasks are not using the processor. The AI server thus isolates the potentially high-variance CROPS5 problem-solving from the real-time tasks. In the CROPS5 architecture, the problem-solving mechanism does not explicitly control the guaranteed real-time tasks. Instead, the production system has separate tasks to perform, and the goal is to ensure that they will also be completed on-time despite running within the best-effort AI server.

Research on CROPS5 has focused on reducing the variance in its processing time, using both enhanced context-switching mechanisms and structuring of the problem space. While performance guarantees have been verified by hand for the system, it does not yet include internal mechanisms for reasoning about its own timeliness or problem-solving capacity. The system does not reason about a model of agent/environment interactions to create its own performance guarantees.

6.1.3 Rex/Gapps

Research into the formal relationship between a system's internal model of the world and the real world has been fruitfully implemented in the Rex/Gapps system [42, 41]. Rex is a language used to describe digital machines that can be viewed as reactive systems. Rex programs are compiled into automata descriptions (usually implemented on a general purpose computer) that perform a constant-time mapping between inputs (sensors) and outputs (actuators). The theory underlying Rex has been used to show that the information stored within a Rex machine can have a fixed relationship to the true state of the world. Thus Rex machines provide predictable execution and support the types of performance guarantees enforced by CIRCA's RTS.

Gapps [21] is a system for compiling declarative descriptions of agent behaviors into Rex machines. Gapps takes as input the agent's top-level goal and a set of goal-reduction rules that describe how to transform goals into smaller goals or Rex-machine primitives. Because Gapps compiles this input into a static Rex machine, it generates large reactive systems that exhibit goal-directed behavior but do not perform lookahead planning, search, or adaptation. Rex/Gapps is used to specify an agent's control mechanisms directly, as in a robot programming language. CIRCA, on the other hand, plans those control mechanisms automatically given a description of goals, primitive capabilities, and the environment.

6.1.4 Any-Time Algorithms

One technique for combining high-variance methods with hard deadlines has recently become popular in both the AI and real-time communities. “Any-time” algorithms [7] are incremental methods that can be interrupted at any time, yielding a result that may have reduced precision, confidence, accuracy, etc. These techniques are naturally successful at making timeliness guarantees: they ensure that some result will be available by a deadline. However, the quality or correctness of that result cannot be guaranteed [32]. Thus any-time algorithms sacrifice correctness for timeliness, while CIRCA strives to guarantee both. Furthermore, by reasoning explicitly about its goals, capabilities, and deadlines, CIRCA can trade off the guarantees it chooses to enforce when constrained by limited resources.

The “imprecise computation” paradigm [24] is a modification of the any-time method in which some minimum amount of processing is guaranteed, so that the algorithm will always produce a result with a minimally acceptable result. This is the technique used by CIRCA in generating TAP plans (see Section 5.1.3), where a minimally acceptable plan achieves only the control-level goals.

6.1.5 PRS

CIRCA's AIS is derived from PRS [14, 20], which itself has some features making it suited to real-time applications. Ingrand and Georgeff have shown that, given certain assumptions about event frequencies and the form of the system's procedural knowledge, PRS can be guaranteed to notice (or begin reacting to) every world event within a bounded time. This guarantee is based on the fact that PRS processing is highly interruptible. However, “noticing” an event is distinguished from responding to the event. PRS does not make guarantees that it will respond to an event by a certain deadline, because it does not (yet) have the ability to reason internally about its own level of reactivity. PRS cannot focus its attention and ignore unnecessary sensor information completely; instead, the world model is constantly updated. Thus the system's response to a particular event can be arbitrarily interrupted by the arrival of other events, and the response to those events can delay the initial processing.

It is possible to limit the system's inferencing capabilities and make guarantees about overall response time [20]. This approach leads to a complete embedding of the AI system within the real-time application environment [34], and requires either low utilization or engineering out the high-variance unpredictability that distinguishes AI techniques from simple algorithms. The guarantees that PRS makes are external to the system's operation.

6.2 Planned Reactions, Proven Safety

As noted earlier, many reactive AI systems have been composed of manually-engineered reaction “plans.” Some systems have been designed with higher-level reasoning processes that select which of the available reactive elements are active [3, 6, 13, 16, 30]. Other reactive systems are designed, like CIRCA, to automatically generate reaction plans from primitive component descriptions [29]. Performing this type of reaction planning is similar to classical planning in the sense that it is done before the plan is executed, and usually involves projecting the effects of proposed reactions in a world model. With such explicit reasoning about the results of plans, it is possible to prove that they will achieve some safety or stability criterion when executed reliably.

6.2.1 Universal Plans

Schoppers' research on the automatic generation of Universal Plans (UPs) [44, 45] resembles our work, with the notable exception that CIRCA relies on a restricted world model and emphasizes timeliness issues. UPs are generated without considering precisely which world states are possible and which are not; UPs specify reactions for *all* states of the world, possible or not. This approach has the advantage that it makes no assumptions about the success of its own actions or the behavior of the external world. However, lacking those assumptions, UPs cannot provide any performance guarantees. CIRCA's control plans can be viewed as "partial Universal Plans," in the sense that they specify reactions, as necessary, for all *possible* worlds. The possibility of a world state, of course, is dependent on the world model assumptions.

We have described how CIRCA's control plans are intended to actively restrict the world to a safely-controlled set of states, maintaining its safety while making progress towards its goals. Schoppers [47] has recently discussed how UPs can similarly lead to stable "closed-loop dynamics." This concept of stable closed-loop control requires that, given sensed data within some bounds (input), the controlled system will produce world behaviors (output) within some bounds. CIRCA reasons explicitly about its ability to meet or alter those bounds, as well as the metric timing information required for guaranteed performance. UPs do not yet handle this type of metric information or the introspective reasoning required to internally verify or alter system goals.

6.2.2 $\mathcal{RS}$

Lyons *et al.* [26, 27] are investigating the Robot Schemas ($\mathcal{RS}$) plan representation with many of the same goals as our work on CIRCA. In the $\mathcal{RS}$ model, robot plans are represented as concurrent communicating processes. $\mathcal{RS}$ provides operators to compose larger systems from various combinations of processes. These composition operators are capable of representing on-line decision-making, concurrent actions, sequential actions, and preconditions. The $\mathcal{RS}$ model can be used to represent both the capabilities of a control system and its environment, just as in CIRCA. Rewrite rules describe the evolution of $\mathcal{RS}$ systems, and these rules can be used to derive proofs that systems will meet their goals [27].

$\mathcal{RS}$ research began by describing static, hand-coded robot control systems. An execution environment is now being developed to allow the system to run its schemas with predictable, guaranteed timeliness [26]. A planning technique has also been proposed [26], in which a concurrent planning process incrementally modifies the reactive schemas running on the execution system.

7 Summary and Future Work

We have described how CIRCA reasons about a principled characterization of agent/environment interactions to generate reactive control plans that are guaranteed to keep the agent safe and, if possible, to drive the system towards its goals. The characterization of agent/environment interactions takes the form of a state-transition model of the world. Borrowing from real-time computing literature, the model includes explicit worst-case timing information that is used to derive the required rate of reaction for various world conditions. By reasoning about this explicit model of the world, CIRCA is also able to recognize when it does not have sufficient resources to guarantee that it will achieve a particular goal within some environment. In that case, CIRCA may choose to leave the goal unguaranteed but still try to achieve it (best effort), or it may alter its high-level plans or goals.

Combining these control-plan generation methods with CIRCA's architectural isolation of the AIS from the RTS yields a system uniquely capable of building and predictably executing control plans that are guaranteed to be timely and correct.

Earlier prototypes of CIRCA controlled a Hero mobile robot navigating through hallways. CIRCA currently controls a simulated robot arm performing the box-packing example discussed throughout this paper. The simulation, written in Deneb Robotics' Igrip system, correctly models the robot's kinematic behavior. However, the prototype real-time subsystem (written in C) runs on a UNIX platform, and thus cannot rigidly enforce execution timing constraints. We are currently porting the RTS to execute on the MARUTI real-time operating system [19], which will provide a predictable execution environment.

Our experience with the current prototype system indicates that the Scheduler module is the weak link, because it uses such a simple algorithm. A more powerful Scheduler, able to take into account dependencies between TAPs (such as precedence and mutual exclusion), would be tremendously helpful, because it would allow CIRCA to schedule and guarantee more TAPs, yielding higher utilization of RTS resources. This improved capacity would, in turn, make it more likely that the system will have sufficient resources for a given problem domain. We are also considering extensions to the Scheduler that will allow it to provide more useful, intelligent feedback to the AIS when resource restrictions prohibit the Scheduler from building a complete TAP schedule.

Acknowledgment

The authors would like to thank Phil Agre, Jim Dolter, Vic Lortz, Melissa Musliner, Jennifer Rexford, Marcel Schoppers, and the anonymous reviewers for their insightful comments on this work. Many thanks are also due to Clare Congdon, Marc Huber, and Mike Hucka for their work on taming Igrip.

References

[1] P. E. Agre and D. Chapman, Pengi: An implementation of a theory of activity, in: Proc. National Conf. on Artificial Intelligence (1987) 268–272.

[2] J. F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (11) (1983) 832–843.

[3] R. C. Arkin, Integrating behavioral, perceptual, and world knowledge in reactive navigation, in: Robotics and Autonomous Systems 6 (1990) 105–122.

[4] R. A. Brooks, A robust layered control system for a mobile robot, IEEE Journal of Robotics and Automation RA-2 (1) (1986) 14–22.

[5] D. Chapman, Planning for conjunctive goals, Artificial Intelligence 32 (3) (1987) 333–374.

[6] J. Connell and P. Viola, Cooperative control of a semi-autonomous mobile robot, in: Proc. IEEE Int'l Conf. on Robotics and Automation (1990) 1118–1121.

[7] T. Dean and M. Boddy, An analysis of time-dependent planning, in: Proc. National Conf. on Artificial Intelligence (1988) 49–54.

[8] T. L. Dean, Intractability and time-dependent planning, in: Proceedings of the 1986 Workshop on Reasoning about Actions & Plans (1987) 245–266.

[9] E. H. Durfee, A cooperative approach to planning for real-time control, in: Proc. Workshop on Innovative Approaches to Planning, Scheduling, and Control (1990) 277–283.

[10] K. Erol, D. Nau, and V. S. Subrahmanian, When is planning decidable?, in: Proc. Int'l Conf. on Artificial Intelligence Planning Systems (1992) 222–227.

[11] R. J. Firby, An investigation into reactive planning in complex domains, in: Proc. National Conf. on Artificial Intelligence (1987) 202–206.

[12] M. K. Franklin and A. Gabrielian, A transformational method for verifying safety properties in real-time systems, in: Proc. Real-Time Systems Symposium (1989) 112–123.

[13] E. Gat, Reliable goal directed reactive control for autonomous mobile robots, Ph.D. thesis, Virginia Polytechnic Institute (1991).

[14] M. P. Georgeff and F. F. Ingrand, Decision-making in an embedded reasoning system, in: Proc. Int'l Joint Conf. on Artificial Intelligence (1989) 972–978.

[15] P. Godefroid and F. Kabanza, An efficient reactive planner for synthesizing reactive plans, in: Proc. National Conf. on Artificial Intelligence (1991) 640–645.

[16] S. Hanks and R. J. Firby, Issues and architectures for planning and execution, in: Proc. Workshop on Innovative Approaches to Planning, Scheduling, and Control (1990) 59–70.

[17] S. Hanks, Practical temporal projection, in: Proc. National Conf. on Artificial Intelligence (1990) .

[18] J. Hendler, Abstraction and reaction, in: Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments (1990) .

[19] J. Hendler and A. Agrawala, Mission critical planning: AI on the MARUTI real-time operating system, in: Proc. Workshop on Innovative Approaches to Planning, Scheduling, and Control (1990) 77–84.

[20] F. F. Ingrand and M. P. Georgeff, Managing deliberation and reasoning in real-time ai systems, in: Proc. Workshop on Innovative Approaches to Planning, Scheduling, and Control (1990) 284–291.

[21] L. P. Kaelbling and S. J. Rosenschein, Action and planning in embedded agents, in: Robotics and Autonomous Systems 6 (1990) 35–48.

[22] Z. Kohavi, Switching and finite automata theory, (McGraw-Hill, New York, 1978).

[23] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, Real-time knowledge-based systems, AI Magazine 9 (1) (1988) 27–45.

[24] K.-J. Lin, S. Natarajan, and J. W.-S. Liu, Imprecise results: Utilizing partial computations in real-time systems, in: Proc. Real-Time Systems Symposium (1987) 210–217.

[25] C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM 20 (1) (1973) 46–61.

[26] D. M. Lyons, A. J. Hendriks, and S. Mehta, Achieving robustness by casting planning as adaptation of a reactive system, in: Proc. IEEE Int'l Conf. on Robotics and Automation (1991) 198–203.

[27] D. M. Lyons, A process-based approach to task plan representation, in: Proc. IEEE Int'l Conf. on Robotics and Automation (1990) 2142–2147.

[28] D. McDermott, A temporal logic for reasoning about processes and plans, Cognitive Science 6 (1982) 101–155.

[29] D. McDermott, Planning reactive behavior: A progress report, in: Proc. Workshop on Innovative Approaches to Planning, Scheduling, and Control (1990) 450–458.

[30] D. P. Miller and E. Gat, Exploiting known topologies to navigate with low-computation sensing, in: Proc. SPIE Sensor Fusion Conf. (1990) .

[31] D. J. Musliner, CIRCA: The cooperative intelligent real-time control architecture, Ph.D. thesis, The University of Michigan, Ann Arbor, MI (1993). Also available as CSE-TR-175-93.

[32] D. J. Musliner, E. H. Durfee, and K. G. Shin, Any-dimension algorithms, in: Proc. Workshop on Real-Time Operating Systems and Software (1992) 78–81.

[33] D. J. Musliner, E. H. Durfee, and K. G. Shin, Reasoning about bounded reactivity to achieve real-time guarantees, in: Working Notes of the AAAI Spring Symp. on Selective Perception (1992) 104–107.

[34] D. J. Musliner, E. H. Durfee, and K. G. Shin, CIRCA: a cooperative intelligent real-time control architecture, IEEE Trans. Systems, Man, and Cybernetics 23 (6) (1993) .

[35] 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 (1994) .

[36] N. J. Nilsson, Principles of artificial intelligence, (Tioga Press, Palo Alto, CA., 1980).

[37] C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider, Reducing problem-solving variance to improve predictability, Communications of the ACM 34 (8) (1991) 81–93.

[38] J. L. Peterson, Petri net theory and the modeling of systems, (Prentice-Hall, Englewood Cliffs, 1981).

[39] J. R. Quinlan, Induction of decision trees, Machine Learning 1 (1986) 81–106.

[40] P. J. G. Ramadge and W. M. Wonham, The control of discrete event systems, Proceedings of the IEEE 77 (1) (1989) 81–98.

[41] S. J. Rosenschein, Synthesizing information-tracking automata from environment descriptions, Technical Report 2, Teleos Research, (1989).

[42] S. J. Rosenschein and L. P. Kaelbling, The synthesis of digital machines with provable epistemic properties, in: Proc. Conf. Theoretical Aspects of Reasoning About Knowledge (1986) 83–98.

[43] S. J. Russell and S. Zilberstein, Composing real-time systems, in: Proc. Int'l Joint Conf. on Artificial Intelligence (1991) 212–217.

[44] M. J. Schoppers, Universal plans for reactive robots in unpredictable environments, in: Proc. Int'l Joint Conf. on Artificial Intelligence (1987) 1039–1046.

[45] M. Schoppers, Automatic synthesis of perception driven discrete event control laws, in: Proc. 5th IEEE Int'l Symposium on Intelligent Control (1990) 410–416.

[46] M. Schoppers, Introduction to special edition on real-time knowledge-based control systems, Communications of the ACM 34 (8) (1991) 27–30.

[47] M. Schoppers, Representing the plan monitoring needs and resources of robotic systems, in: Proc. Annual Conf. on AI, Simulation, and Planning in High Autonomy Systems (1992) .

[48] H. A. Simon, Models of bounded rationality, (M. I. T. Press, 1982).

[49] J. A. Stankovic, Misconceptions about real-time computing: A serious problem for next-generation systems, IEEE Computer 21 (10) (1988) 10–19.

[50] S. Vere, Temporal scope of assertions and window cutoff, in: Proc. Int'l Joint Conf. on Artificial Intelligence (1985) 1055–1059.