ABSTRACT

CIRCA:
THE COOPERATIVE INTELLIGENT REAL-TIME CONTROL ARCHITECTURE

by
David John Musliner

Co-Chairs: Kang G. Shin and Edmund H. Durfee

The Cooperative Intelligent Real-time Control Architecture (CIRCA) is a novel architecture for intelligent real-time control that can *guarantee* to meet hard deadlines while still using unpredictable, unrestricted AI methods. CIRCA includes a real-time subsystem used to execute reactive control plans that are guaranteed to meet the domain's real-time deadlines, keeping the system safe. At the same time, CIRCA's AI subsystem performs higher-level reasoning about the domain and the system's goals and capabilities, to develop future reactive control plans. CIRCA thus aims to be *intelligent <u>about</u> real-time*: rather than requiring the system's AI methods to meet deadlines, CIRCA isolates its reasoning about which time-critical reactions to guarantee from the actual execution of the selected reactions.

The formal basis for CIRCA's performance guarantees is a state-based world model of agent/environment interactions. Borrowing approaches from real-time systems research, the world model provides the information required to make real-time performance guarantees, but avoids unnecessary complexity. Using the world model, the AI subsystem develops reactive control plans that restrict the world to a limited set of safe and desirable states, by guaranteeing the timely performance of actions to preempt transitions that lead out of the set of states. By executing such "safe" and "stable" plans, CIRCA's real-time subsystem is able to keep the system safe (in the world as modeled) for an indeterminate amount of time, while the parallel AI subsystem develops the next appropriate plan.

We have applied a prototype CIRCA implementation to a simulated Puma robot arm performing multiple tasks with real-time deadlines, such as packing parts off a conveyor belt and responding to asynchronous interrupts. Our experimental results show how the system can guarantee to accomplish these tasks under a given set of domain conditions (e.g., conveyor belt speed) and resource limitations (e.g., robot arm speed). Furthermore, because CIRCA reasons explicitly about its own predictable, guaranteed behaviors, the system can recognize when its resources are insufficient for the desired behaviors (e.g., parts are arriving too quickly to be packed carefully), and can then make principled modifications to its performance (e.g., temporarily stacking parts on a table) to maintain system safety.

CIRCA:
THE COOPERATIVE INTELLIGENT REAL-TIME CONTROL ARCHITECTURE

by

David John Musliner

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1993

Doctoral Committee:
Professor Kang G. Shin, Co-Chair
Assistant Professor Edmund H. Durfee, Co-Chair
Associate Professor Daniel Koditschek
Associate Professor John E. Laird
Associate Research Scientist Terry E. Weymouth

To Melissa.

ACKNOWLEDGEMENTS

I am most grateful to my two graduate advisors for their continuous encouragement and guidance. Professor Shin has provided a strong base of funding, enthusiasm, the lore (lure?) of real-time systems principles, and a tempered skepticism of AI that has kept my feet on the ground. Professor Durfee has provided a virtual fountain of excitement and ideas about AI and real-time, and an unparalleled ability to see the better way of doing things. Many new graduate students have asked me whether having two advisors is a good idea; my response has always been that, between the two, I have received twice the beneficial effects of excellent advising: guidance, ideas, assistance, and, above all, encouragement.

I would also like to gratefully acknowledge:

- The other members of my dissertation committee, for their constructive suggestions and feedback. Professor Koditschek deserves particular thanks for going beyond the call of duty, flying back to Ann Arbor for the defense.
- My wife Melissa, for her love, support, and patience.
- My parents, for instilling in me a love of learning, a curious mind, and an engineering spirit.
- Jim Dolter, for five years of sharing office space and computing know-how. Jim has been a great friend and a willing listener, and has provided numerous ideas and a great deal of technical help with all aspects of my work.
- Jennifer Rexford, for reading draft material and for several excellent ideas about examples.
- My other officemates, for living with me even when I brought in delicious-smelling food for lunch.
- Phil Agre and Marcel Schoppers, for helpful suggestions and discussions.
- The National Science Foundation and the Rackham Graduate School, for fellowship and grant funding (IRI–9209031 and IRI–9158473) that allowed me to focus on research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

LIST OF APPENDICES

APPENDIX

CHAPTER 1

INTRODUCTION

The conceptual goal of this dissertation is, quite simply, the combination of two major areas of Computer Science research: real-time computing and Artificial Intelligence (AI). In discussing this concept, we will find that these two research areas are pursuing conflicting goals; while real-time computing is aimed at producing predictable systems that make fixed performance guarantees given limited resources, AI is trying to produce flexible systems that behave "intelligently" in complex, dynamic domains. We will explore the nature of these conflicting goals and present several alternative methods for avoiding the conflicts and merging real-time and AI research. This discussion will culminate in the introduction of a new architecture designed to combine real-time computing and AI. The Cooperative Intelligent Real-time Control Architecture (CIRCA) provides a powerful, flexible combination of real-time and AI capabilities, meeting goals that current systems have not addressed. To investigate the strengths and limitations of CIRCA, we will describe a prototype implementation in great detail, and illustrate the system's operations with examples from several domains.

1.1 Motivation

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 dynamic 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 also 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 [36, 76].

This understanding of what it means to be "real-time" is dramatically different from the casual, non-technical use of the term which has become common in many fields. For example, if a database querying system responds quickly according to human time-scales (i.e., in a few seconds or less), it is called "real-time." But what if we use that same database system in a critical application requiring responses in milliseconds? Clearly, the system is no longer "fast enough." The fact that the inadequacy of the system in this new domain (and its "adequacy" in the slower domain) could not be recognized or predicted in any rigorous fashion indicates that this system was never "real-time" in the technical sense; it was never known to meet the required deadlines.

Real-time systems researchers have developed a powerful set of tools to prove that embedded systems meet their environment's deadlines. 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 consider 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 actions (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.

Note that human behavior is not a "gold standard" for real-time systems research. Although much AI research seeks to emulate human performance, people do not fully meet the criterion for real-time systems. In particular, human performance is too uncertain and too subject to unpredictable delays, distractions, and errors to be considered "guaranteed" in the sense needed for hard real-time domains. It is certainly true that humans can perform in demanding environments that include deadlines and response requirements: people drive cars, fly planes, etc. However, the high incidence rate of car accidents can be largely attributed to human error— people are simply not predictable enough to reliably handle the task. Our research takes a first step towards combining the best aspects of intelligent, human behavior (flexibility, adaptability, robustness) with the strengths of real-time computing systems (predictability, performance guarantees, reliability).

Unfortunately, applying the insights of real-time computing to the development of intelligent embedded agents is not trivial. One major problem in combining AI and real-time systems is complexity. A real-time system must be guaranteed to meet the hard deadlines that its environment imposes, even under its worst-case performance. This requirement is

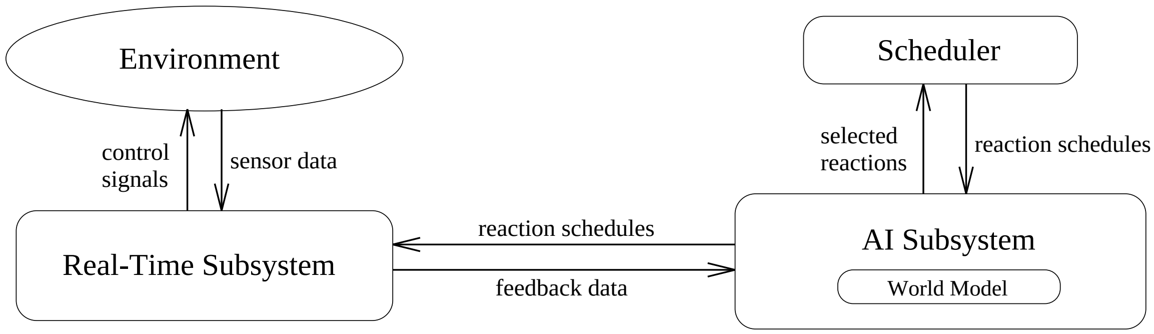


Figure 1.1: The Cooperative Intelligent Real-time Control Architecture (CIRCA).

difficult for intelligent systems because many AI techniques are not suited to analyses that can provide worst-case response times. For example, systems that learn are able to form new chains of inferences, resulting in changing performance characteristics that may defy worst-case bounding [13]. Furthermore, even predictable AI methods often have such high variance in their response times that making guarantees based on worst-case values would result in severe underutilization of computational resources during normal operations [60]. Thus making real-time performance guarantees while still using complex AI methods is a fundamental problem.

1.2 Approach and Contributions

In pursuit of the overall goal of combining real-time computing methods and AI techniques, this dissertation focuses primarily on describing the concepts underlying CIRCA, and on the particular implementation we have developed. As illustrated in Figure 1.1, 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 with the Scheduler, develops low-level control plans consisting of reactive behaviors. These control plans are executed in a predictable, guaranteed fashion by the real-time subsystem (RTS).

Unlike many other real-time AI systems, which force their AI components to meet real-time deadlines, CIRCA explicitly isolates its AIS from domain deadlines. CIRCA's goal is to be "intelligent <u>about</u> real-time," rather than "intelligent <u>in</u> real-time." This distinction is crucial because it allows CIRCA to provide performance guarantees that are distinctly different from those available with other systems. While many real-time AI systems can only promise "best-effort" performance, CIRCA is able to make explicit guarantees about its ability to achieve its goals within particular domains using limited sensor, processor, and actuator resources.

In a sense, the goal of this research is to automate the design, implementation, and modification of a real-time control system. Currently, real-time systems are built by human designers who are given a set of task specifications, and design a method for executing those tasks to meet the specified constraints. CIRCA is designed to receive a description of its environment, its capabilities for interacting with its environment, and its goals, and then automatically derive a behavioral plan, confirm that the system has sufficient resources to implement that plan (or modify the plan, goals, or environment as necessary), and finally implement the plan.

In pursuit of these objectives for CIRCA, this dissertation makes several conceptual contributions to the state of the art:

- To clarify the fundamental conflict between real-time and AI systems, we introduce the concept of *any-dimension algorithms*, a general class of iterative improvement algorithms.
- To allow CIRCA to build real-time reactive plans, we develop a graph-based world model for representing the interactions between the environment and CIRCA's real-time subsystem. The model includes a simplified temporal representation that permits easy analysis of worst-case behaviors.
- Within the world model representation, we develop a formal characterization of the way a reactive plan can isolate the AI subsystem from the domain deadlines, keeping the controlled agent safe while CIRCA's AI subsystem executes unpredictable algorithms.
- Using these capabilities, we characterize the domains to which CIRCA may usefully be applied, examining the range of situations in which our approach to real-time AI is appropriate.

Our research on actually implementing a prototype version of CIRCA has yielded several technical contributions, including:

- A novel deliberative architecture combining meta-level reasoning with interrupt-driven communication.
- A structured interface through which the arbitrarily complex AI planning subsystem can communicate with and control a predictable, guaranteed real-time subsystem.
- A real-time subsystem that meets hard response deadlines by executing a cyclic schedule of behaviors.
- A modular, interruptible, search-based algorithm that the AIS uses to plan within the world model, determining which behaviors to request from the real-time subsystem.
- A modified deadline-driven scheduling algorithm that efficiently produces cyclic schedules of reactive behaviors.
- A set of methods by which the AIS can modify its world model or plans to decrease the resource requirements of the reactive behaviors it is trying to schedule for the real-time subsystem.

The prototype CIRCA implementation has exhibited several forms of novel performance. Primarily, the system has demonstrated the feasibility of using unpredictable AI algorithms in the process of building and executing reaction plans that can provide rigorous real-time guarantees. Other experiments have concentrated on showing the system's unusual combination of introspection and real-time performance. CIRCA is able to recognize its sensing, actuating, and processing resource limitations, and make performance tradeoffs when those resources do not allow the system to meet all of its goals within a given environment. Experiments have also been conducted to investigate the use of unguaranteed (or "best-effort") behaviors that take advantage of the execution resources that become available when guaranteed behaviors use less than their worst-case times. CIRCA supports cognizant, graceful performance degradation via these best-effort behaviors and tradeoff methods.

1.3 The Example Domain

Throughout this dissertation, we will discuss examples drawn from the domain shown in Figure 1.2. The Puma robot arm is simulated in Deneb Robotics' Igrip system (see Appendix A for details). The Puma is assigned the task of packing parts arriving on the conveyor belt into the nearby box. The conveyor moves at a fixed rate and the 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 belt because the robot does not pick it up in time, the system is considered to have failed. Thus, the arriving parts impose hard deadlines on the robot's responses; it must always pick up parts before they fall off the conveyor.

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 derivation of the packing method may involve search algorithms with unpredictable behavior. This aspect of the domain is used to exercise CIRCA's ability to combine arbitrary AI methods with real-time response guarantees.

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

To cope with this domain properly, the system controlling the robot arm must be able to provide real-time responses to unsynchronized domain events (part arrivals and emergency alerts) while also having the ability to perform complex search methods (deriving packing methods and reaction plans in general). To complicate matters further, the speed of the Puma robot and the domain sensors is limited. Variations of the domain can be set up with different part arrival rates, emergency alert rates, robot speeds, etc. To be truly intelligent and real-time in this domain, the control system will need to be able to evaluate its capabilities, its goals, and the domain behavior restrictions. With that information, an intelligent system should provide some measure of useful performance, possibly involving tradeoffs that sacrifice aspects of the system behavior as necessitated by resource restrictions.

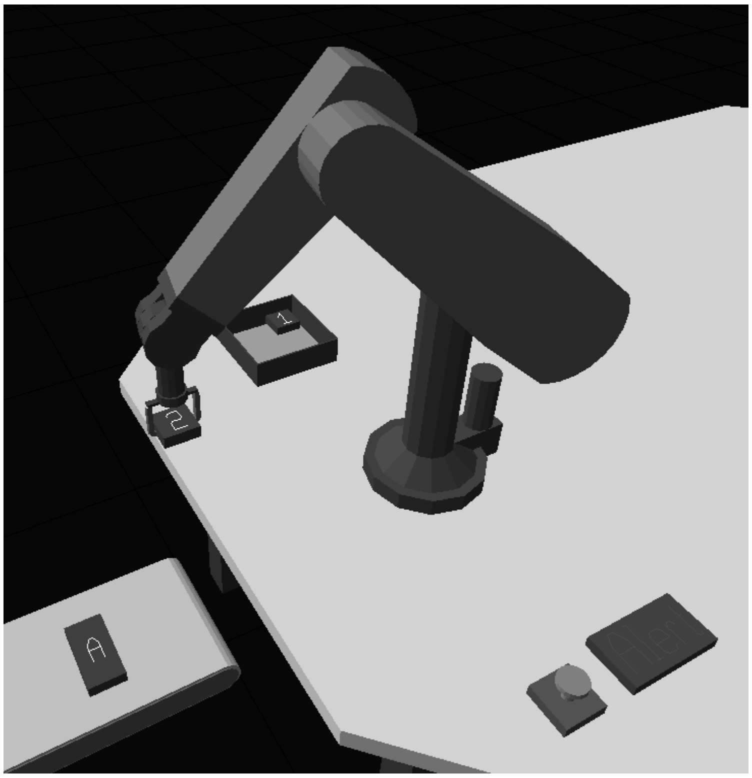


Figure 1.2: The example Puma domain, in which the robot packs objects from the conveyor into the box.

1.4 Dissertation Outline

The body of this dissertation consists of seven additional chapters and several appendices. The chapters follow a logical progression from background concepts (Chapter 2), through an overview of the system (Chapter 3) and its theoretical underpinnings (Chapter 4), into implementation details (Chapters 5 and 6). Experimental results follow in Chapter 7, and the dissertation concludes with a brief review and discussion of future work in Chapter 8. In somewhat more detail, the chapters are organized as follows:

Chapter 2 presents background material describing the differing goals of real-time systems and AI, and shows how combining these research fields is non-trivial. We introduce the concept of *any-dimension algorithms*, which can be used to describe the tradeoffs necessary to combine the disparate goals of real-time and AI systems. This discussion clarifies the problems with many approaches to real-time AI, and leads directly to the conceptual basis of CIRCA.

Chapter 3 presents an overview of CIRCA. We describe the architecture in terms of the division of functional responsibilities among subsystems, and we motivate these divisions based on the goals described above and in Chapter 2. We then discuss in detail the previous approaches to combining real-time and AI, and we show how CIRCA fills a unique gap in the space of possible real-time AI systems.

Chapter 4 provides a detailed description of the world modeling methods CIRCA uses to reason about its environment and its own capabilities. Based on this world model, CIRCA makes decisions about its own performance, trading off various measures of performance quality when it recognizes that its resources are constrained. The final section of this chapter evaluates several advantages and disadvantages of the model.

In Chapter 5 and Chapter 6, we provide extensive details on the current prototype implementation of CIRCA. The implementation includes several novel features, including a reaction planner with a simplified representation of time, and a programmable real-time subsystem with predictable performance. Each of the prototype CIRCA subsystems is briefly evaluated according to its strengths and weaknesses.

Chapter 7 returns to the any-dimension algorithm theme to guide an evaluation of the performance of the CIRCA implementation, focusing particularly on the performance tradeoffs the system can make. While the implementation is less well-developed than several older architectures, we are able to demonstrate several unique performance features that result from CIRCA's innovative approach to combining real-time and AI.

The dissertation concludes with Chapter 8, which reviews the contributions of CIRCA and discusses interesting directions for future work. Several appendices provide additional details on features of the implementation and the testing domains.

CHAPTER 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, on the other hand, are concerned with proving that the time needed by a set of actions will not exceed deadlines. Ideally, we would like to combine 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.

In this chapter, we will discuss alternative ways of meeting real-time constraints, and reveal a fundamental conflict between these methods and the characteristics of traditional AI methods. To clarify the nature of this conflict, we will introduce any-dimension algorithms, a generalized notion of iterative computation. The any-dimension concept will allow us to precisely pinpoint the conflict between the goals of real-time system and AI systems. With that understanding, we then survey several approaches to resolving this conflict, before introducing our approach in Chapter 3.

2.1 The Strategic Approach to Real-Time Guarantees

As noted in Chapter 1, 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. There are two main classes of methods for obtaining performance guarantees given a limited set of system resources. In the most common "strategic" approach, a scheduler is given information about resource availability and future computational tasks, and determines how to execute those tasks in order to avoid resource conflicts and meet some performance requirements. This approach is well-suited to simple control algorithms and static domains, where resource needs and availability are predictable, so that the resulting schedule of tasks can be followed precisely.

Unfortunately, the search-based AI methods used in complex planning systems are problematic for strategic schedulers. The fundamental problem is that planning involves searching for the solution to a generally intractable problem [7], 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 of a planner will be very costly, and will lead to very low utilization of a system's resources [60, 68]. Furthermore, AI systems with powerful knowledge representations [7, 14] or learning abilities [13] 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. As a result, it is not generally possible to build an effective real-time AI system by embedding traditional AI methods within a real-time system using the strategic approach to response guarantees.

2.2 The Tactical Approach to Real-Time Guarantees

To avoid these problems of strategic scheduling, some researchers have focused on "tactical" approaches that rely on the computational tasks themselves to manage their resource usage. These tactical methods are exemplified by *any-time algorithms* [9, 41, 65], which can be halted at any time to yield a result, possibly with reduced precision, confidence, or completeness. Any-time algorithms provide an on-line, dynamic method for guaranteeing the timeliness of a result, but the quality of the result may be sacrificed. For example, in the Puma domain, an any-time algorithm might be used to incrementally refine the robot's estimate of the position of a part arriving on the conveyor belt. When the time allotted to the any-time algorithm expires, another process could revise the robot's motion based on the resulting position estimate. However, the accuracy of the position estimate would be highly dependent on how much time was actually allocated to the any-time algorithm.

To clarify the relationship between AI and the tactical real-time systems approach, it is helpful to focus closely on tactical systems as implemented by *any-dimension algorithms*. The any-dimension algorithm concept is simply a generalization of the any-time approach, yielding a description of a larger class of tactical algorithms that can provide performance guarantees along dimensions describing resource usage and/or solution quality, rather than just time. Note that we are not claiming that any-dimension algorithms are a new method, but rather that this particular form of algorithm *description* is useful. The following subsections develop the concept in detail, showing how both real-time and AI methods can be mapped into any-dimension algorithms. We will use the any-dimension concept to our advantage in Section 2.3, where it will provide leverage on both describing and attacking the problems of combining real-time and AI.

2.2.1 Any-Dimension Algorithms

The most important feature of an any-time algorithm is the fact that it guarantees to use only a bounded amount of time, by performing iterative computations that can return a result any time they are halted. Of course, time is not the only resource that may be limited for a system: other bounded resources might include memory and non-computational physical features like sensors and actuators. We can generalize the any-time concept to provide guarantees on other dimensions by noting that any-time algorithms have two crucial elements: an iterative computation that produces intermediate results, and a termination condition that monitors the time and halts the iteration when the deadline is reached. So an *any-dimension* algorithm is composed similarly of an iterative computation and a ter-

```
do
   {
      new_result = compute_next_result_from(best_result_so_far);
      best_result_so_far = best_of(best_result_so_far, new_result);
   }
until termination_condition();     /* threshold on resource/quality */
return(best_result_so_far);
```

Figure 2.1: Pseudo-code for a generic any-dimension algorithm.

mination condition that may keep track of any measure of an algorithm's performance, and will halt the iterative computation when some threshold on that measurement dimension is reached. Figure 2.1 illustrates a generic any-dimension algorithm in pseudo-code.

There are two basic types of any-dimension algorithms, distinguished by the nature of their termination conditions. These conditions may perform thresholding tests on measures of either resource usage or output quality.

2.2.2 Any-Resource Algorithms

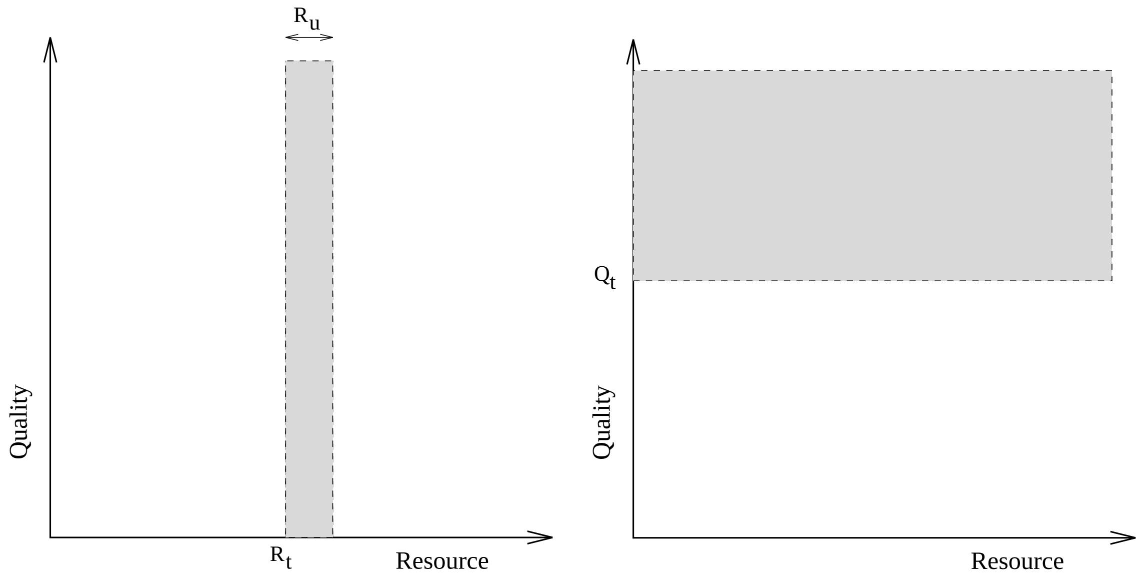
Any-resource algorithms are the most obvious generalization of any-time algorithms, having a termination condition that tests for some maximum resource usage:

```
boolean termination_condition ()
{
  return(resources_used >= max_resource_threshold);
}
```

Any-resource algorithms can guarantee that they will not exceed a maximum level of resource usage. As another example of an any-resource method, consider a scenario in the Puma domain where the system's planning process has no hard deadline, but the system has limited memory. An any-memory algorithm would be useful in this situation, because the planning algorithm could require exponential amounts of memory as it constructs and stores alternative partial plans. If the planner writes beyond the free memory, it might corrupt critical control data and cause a catastrophic failure. An any-memory planning algorithm would monitor the available memory and, when memory ran low, the algorithm would halt and return the most-recent partial plan. Thus an any-memory algorithm guarantees that the system will not exceed the available memory capacity.

We can qualitatively represent the results of this type of algorithm by the resource/quality tradeoff graph in Figure 2.2a, where the shaded area represents the possible places that the any-resource algorithm will terminate (i.e., the types of results it will produce). Because we have cast the termination condition as a synchronous monitor within the iterative loop, it will only check the resource usage after each iteration[1]. As a result, the algorithm may overshoot the resource threshold (R_t) by up to the maximum amount of resources used

[1] In this discussion, we are not concerned with alternative, asynchronously-monitored termination conditions that might rely on interrupts to halt the iterative computation [55].



(a) Simple any-resource algorithm. (b) Simple any-quality algorithm.

Figure 2.2: Termination regions for simple any-dimension algorithms.

during any single iteration of the computation (R_u). Thus Figure 2.2a shows that a simple any-resource computation will be halted at some point when the resource usage is between R_t and $R_t + R_u$.

Note that an any-resource algorithm may fail to terminate if it never consumes enough resources. While this is not possible if the resource dimension is time (and there is a finite threshold, or deadline), with other resources it is possible for an iterative algorithm to continue executing without consuming additional resources. For example, a simple beam-search algorithm may traverse an arbitrarily large search space with a fixed maximum memory usage, and thus, if cast as an any-memory algorithm, it might never terminate.

Unlike strategic scheduling methods, which must be given information about the total available resources and resource requirements, any-resource algorithms can make performance guarantees even when resource limits and needs are changing or unknown when the algorithm starts. Any-resource algorithms dynamically adjust their resource usage to avoid exceeding some maximum level that may be determined outside of the algorithm. Thus any-resource algorithms are particularly appropriate for tasks where multiple computations may be competing for resources; the any-resource algorithms will automatically avoid overtaxing resources. Any-resource algorithms, and particularly any-time algorithms, are suited to real-time domains because they can provide flexible computations guaranteed to meet deadlines and other resource limitations. Unfortunately, any-resource algorithms do not provide any control over their output quality; whenever an any-resource algorithm's resource threshold is reached, it returns the current result, which may have less-than-optimal precision, confidence, completeness, or other quality measures. If result quality is critical, any-resource algorithms are inappropriate.

2.2.3 Any-Quality Algorithms

Any-quality algorithms *can* make output quality guarantees. While similar to an any-resource algorithms in that they iteratively compute intermediate results, any-quality algorithms differ in that their termination conditions are specified by a desired minimum level of result quality, rather than a maximum level of resource availability:

```
boolean termination_condition ()
{
  return(quality(best_result_so_far) >= min_quality_threshold);
}
```

Figure 2.2b shows the termination region for the resulting any-quality algorithm. As with any-resource algorithms, an any-quality algorithm may never terminate if the performance profile of the iterative computation never crosses the quality threshold (Q_t). This observation clarifies the value of "monotonic-improvement" any-quality algorithms: if the iterative computation always improves its result quality, then the iteration can be guaranteed to terminate for any finite Q_t. As a specific example, many iterative numerical methods [6] are any-precision algorithms. An iterative numerical method continually refines its estimate for the solution to a problem until the precision of its estimate is known to be beyond a certain level. In the Puma domain, such an iterative method might also be used to refine the estimate of an arriving part's position until its precision reaches some fraction of a centimeter. The algorithm would continue running until it achieved that level of accuracy, as opposed to the any-time methods discussed above, which terminate when a deadline is reached. If absolute part locations are critical to the robot's task, then an any-precision algorithm would be appropriate, while if the task has hard deadlines, an any-time algorithm might be better. In general, while any-resource algorithms match the goals of resource-constrained real-time systems, any-quality algorithms match the satisficing behavior of many AI methods.

Just as any-resource algorithms cannot guarantee output quality, a fundamental weakness of any-quality algorithms is that they cannot guarantee limited resource usage. By definition, any-quality algorithms must consume resources until they achieve the desired quality threshold.

2.2.4 Combinations of Any-Dimension Algorithms

We have noted that a simple any-dimension algorithm has the disadvantage of being unable to control its performance along more than the single dimension specified in its termination conditions. One approach to fixing this weakness is to combine multiple termination conditions using conjunction and disjunction to yield more interesting algorithmic behavior. Disjunctive (OR) combinations of any-dimension methods lead to a guarantee that crossing one threshold or the other will yield a result. For example, in the Puma domain, combining any-time and any-confidence conditions might be the most appropriate method for building plans to deal with various types of parts under time pressure; the resulting algorithm would work on each planning problem until it either found a result in which it had sufficient confidence, or until the time allotted to that problem expired.

Disjunctive combinations of thresholds are actually quite common. A simple any-quality algorithm will run until its result reaches the quality threshold; if the threshold is too high, the any-quality algorithm may never terminate. Thus, most implementations of any-quality algorithms also include an alternative, resource-based termination condition, so that they will terminate even if their original quality threshold is never reached. For example, an any-precision algorithm might also have a condition that will terminate the algorithm after a certain number of iterations, regardless of the precision that has been reached at that time.

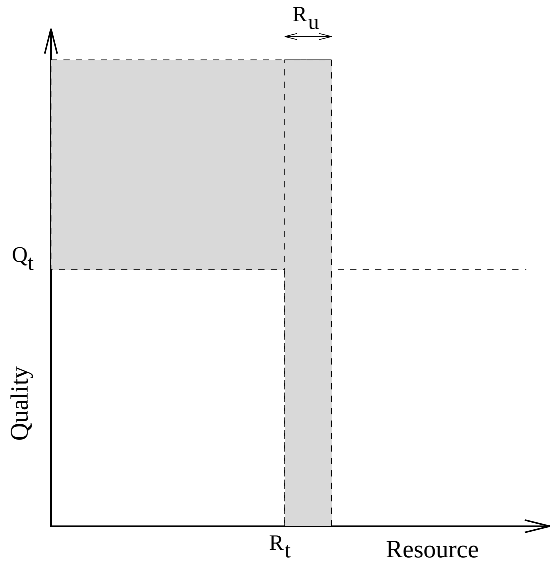
Similarly, a simple any-resource algorithm will run until it has consumed the allocated resources, even if the algorithm finds an optimal (highest quality) result before the resources are exhausted. To avoid this waste of resources, most any-resource algorithms also include a termination condition specifying an acceptable quality measure. For example, a search algorithm might have a termination condition checking for both a deadline and for the goal of the search. If the goal is reached before the deadline, the algorithm terminates and returns its result even though it could have used more time.

In general, a disjunctive combination of an any-quality and an any-resource algorithm can be implemented using the form:

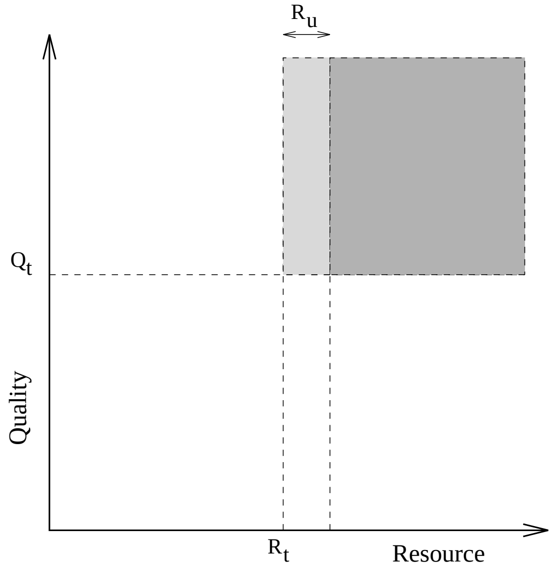
```
boolean termination_condition ()
{
  return( (quality(best_result_so_far) >= min_quality_threshold) ||
          (resources_used >= max_resource_threshold) );
}
```

The termination graph for this disjunctive form is just the union of the previous two graphs, as illustrated in Figure 2.3a. The algorithm will return a result either when it achieves sufficient quality or when it reaches the band of maximum allowable resource usage. This form of algorithm will not fail to terminate as long as some progress is being made along either the quality or resource dimension. This is the reason that disjunctive combinations are quite common: they will definitely terminate, and they provide intuitively desirable results— they continue running until they achieve a result of sufficient quality, or until their resources are consumed, whichever comes first. Note that the decision as to which termination condition (quality or resource) will be the deciding factor is not made until the algorithm actually runs; thus, this is not a simple prioritization mechanism.

The notion of conjunctive (AND) combinations of any-dimension methods is also desirable in certain cases, because it leads to guarantees over multiple dimensions. For example, combining any-confidence and any-precision conditions would lead to results with guaranteed precision and confidence: the algorithm would continue until <u>both</u> thresholds are reached. Given the mapping described earlier between different any-dimension methods and the goals of AI and real-time systems, such conjunctive combinations might seem to point the way to a unified approach to real-time AI. However, if we try to conjoin termination conditions on both resource and quality dimensions, the results are not clearly defined. A conjunctive combination of an any-quality and an any-resource algorithm can



(a) A disjunctive combination.



(b) A conjunctive combination.

Figure 2.3: Termination regions for combined forms of any-dimension algorithms.

be implemented using the form:

```
boolean termination_condition ()
{
  return( (quality(best_result_so_far) >= min_quality_threshold) &&
          (resources_used >= max_resource_threshold) );
}
```

This conjunction results in the undesirable termination pattern shown in Figure 2.3b. The conjunctive combination algorithm terminates only if the desired level of quality is achieved at the same time the resource consumption reaches the specified limits. A conjunctive algorithm would never terminate if it reached and exceeded its quality threshold but never used up the threshold quantity of resources (i.e., if the performance profile shoots up but never crosses R_t). In this case, the results are similar to the behavior of a pure any-resource algorithm: it runs until the resource bound is reached. Likewise, and perhaps even worse, the conjunctive algorithm may also never terminate if the algorithm uses resources beyond the threshold R_t before the quality bound is reached (i.e., the performance profile continues to the right, even past $R_t + R_u$, while remaining below Q_t). The problem with this case is that the algorithm does not terminate even though it has utilized all of the allocated resources— this might lead to unexpected failures, as the algorithm tries to continue using resources. The algorithm will only terminate when it has both achieved sufficient quality <u>and</u> used up all the allocated resources.

This type of conjunction is problematic in realistic systems, because the region beyond the $R_t + R_u$ boundary is ill-defined. For example, a conjunction of any-time and any-precision algorithms will not necessarily obtain both guaranteed precision and guaranteed timeliness. What happens if the time threshold (deadline) is reached before the precision threshold? The deadline indicates that all the allocated resource (time) has been consumed. If the algorithm terminates it fails to achieve the desired precision, but if it continues it will violate the resource threshold. Thus there is a fundamental restriction on conjunctive combinations: they cannot be applied to any-resource algorithms, because resource thresholds represent maxima.

2.3 Any-Dimension Algorithms and Real-Time AI

The inability to build conjunctions of any-resource and any-quality algorithms is at the heart of why real-time AI is so elusive. Real-time systems require resource-usage guarantees; they must produce a result "by the right time." AI, on the other hand, is concerned with solution quality: a chess program should make *good* moves, an autonomous vehicle should turn in the *correct* direction to avoid a collision, etc. So AI systems are designed to "do the right thing[2]." Together, real-time AI systems must "do the right thing, by the right time."

But we have shown that, with tactical any-dimension algorithms, guarantees on resource usage and output quality cannot simply be conjoined. The only way around this problem is to alter the termination condition so that it is no longer in an unacceptable form. One approach to doing this is to map one dimension threshold onto another, reducing the conjunctive any-dimension algorithm to testing a single dimension. For example, if we can convert a termination condition expressed in a quality dimension into an equivalent *minimum* level of resource usage, then we know that reaching the minimum resource threshold will ensure also passing the minimum quality threshold. Figure 2.4 illustrates this mapping operation. Note, however, that now we not only have our usual maximum resource threshold for the any-resource algorithm, but we also have a minimum resource threshold to capture the any-quality dimension. In Figure 2.4, the algorithm's termination must be restricted to the shaded area. To meet the quality requirement, we have to guarantee that at least the minimum quantity of resources will be available for the algorithm. Unfortunately, a tactical any-dimension algorithm cannot make such a guarantee.

However, recall that strategic approaches *can* guarantee resource availability by scheduling tasks before they run. The next logical step, then, is to try to combine the advantageous features of both strategic and tactical methods to yield a new, combined approach that successfully addresses the requirements of real-time intelligent control in dynamic domains.

Given an any-resource algorithm with both minimum and maximum resource requirements, one approach is to use a strategic method to schedule enough of the resource to assure the minimum threshold, and then to employ a tactical method beyond that to dynamically take advantage of additional resources at runtime. This is essentially the approach taken by Liu *et al.* [43] in the "imprecise computation" method. In this paradigm, an algorithm is divided into mandatory computations that are required to reach a minimal quality threshold, and optional computations that incrementally improve the result and can be interrupted at any time. The imprecise computation scheduler builds schedules that allocate at least enough time for all the mandatory computations. Excess time is scheduled for optional

[2] By "the right thing," we mean the best choice given the system's limited knowledge and resources.

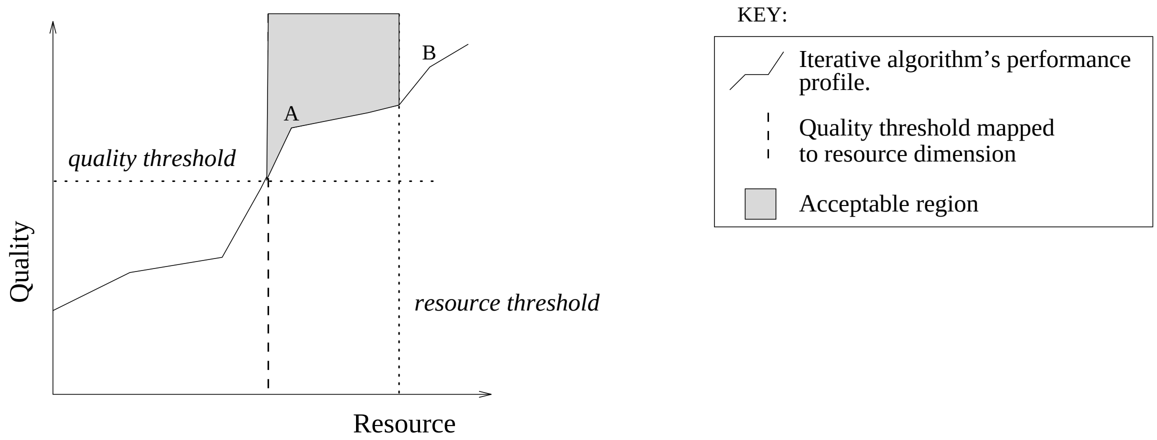


Figure 2.4: An example performance profile, showing how a quality threshold can be mapped to a minimum resource threshold.

computations.

While the imprecise computation approach has the advantage of balancing strategic and tactical considerations to assure minimum quality within resource bounds, it does not provide any method for dealing with the problems that arise when resources are so scarce that all mandatory computations cannot be scheduled. In this over-constrained situation, an intelligent system must make tradeoffs between the level of output quality it will guarantee and the resource usage it schedules. For example, the system might use load-shedding methods [19, 35, 44] to drop or postpone some mandatory task, leaving resources available for the rest. Or, if alternative methods are available for accomplishing a particular task, the system might attempt to schedule lower-cost methods that will produce a lower-quality solution [19, 47]. When making these tradeoffs between solution quality and resource usage, an intelligent system should use principled methods to decide what it will accomplish.

One approach to dealing with over-constrained systems is to make no guarantees of minimum quality, but instead strive to perform "as well as possible" with the given resources. Dean and Boddy's work on "deliberation scheduling" [9] uses decision-theoretic methods to build task schedules that optimize a measure of overall system utility (output quality). The various problem-solving methods that a system might need to run in some situation are cast as any-time algorithms. The deliberation scheduling problem is then to decide how long each competing any-time algorithm should be run. Dean and Boddy assume that a performance profile, like the one in Figure 2.4, is available for each system task, and that these tasks are interruptible, restartable, and completely independent, so that the total system utility is simply the sum of the utility levels achieved by individual tasks. Given these assumptions, a scheduling algorithm can maximize system utility by running, at each moment, the task with the largest expected gain in utility. In over-constrained systems, the any-time algorithms will continue to guarantee output timeliness, but output quality will be sacrificed as much as necessary to meet the deadline.

Thus, while imprecise computation assures minimum solution quality given minimal

```
xguess = initial_xguess;
while (abs(xnew - xguess) > .01)
      {
         xguess = xnew;
         xnew = xguess - F(xguess) / Fprime(xguess);
      }
```

(a) Newton's method.

F	initial_xguess 1	10	20
x^2	7	10	11
$e^x - 1$	4	13	23
$e^{25x} - 1$	27	252	502

(b) Iterations to achieve .01 precision.

Figure 2.5: Showing the difficulty of mapping precision to time for Newton's root-finding method.

resources, deliberation scheduling commits to doing as well as it can given no assumptions on resources. Both approaches assume that the system is given a fixed mapping between the output quality (utility) dimension and the resource usage (time) dimension. There are two fundamental problems with this assumption. First, such mappings may be difficult or impossible to derive, because the performance of most algorithms is highly dependent on the particular problem to which the algorithm is being applied. For example, Liu *et al.* [43] describe an any-time implementation of Newton's method for finding the roots of a function F. Unfortunately, as illustrated in Figure 2.5, the number of iterations this method requires to achieve a result with specified precision is highly dependent on both the domain (the function F) and the internal state of the system (the initial guess for the root value). Because the precision threshold cannot be mapped onto the time dimension, the root-finding computation cannot be cleanly separated into mandatory and optional parts based on time alone.

The second major difficulty is that, even if an individual algorithm's output quality can be accurately characterized by a fixed performance profile, tasks are not independent in realistic domains; the utility of a particular computation depends on other task computations. In the Puma domain, the utility of running a computation to decide whether to pick up a part is dependent on whether the routine that locates parts has been run. Furthermore, the utility of the part-locating routine is affected by the fact that its results will be used to decide about moving the robot. These routines have high utility when used in conjunction, in a particular order, but low utility otherwise.

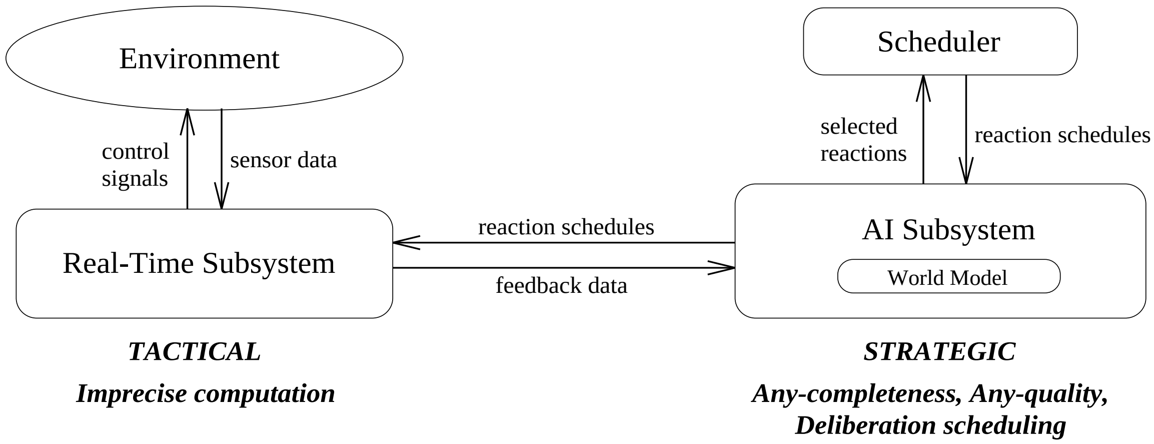


Figure 2.6: CIRCA revisited: combining strategic and tactical methods.

2.4 CIRCA Revisited

The CIRCA approach to real-time AI combines features of both strategic and tactical methods, as shown in Figure 2.6. Together, the AIS and Scheduler act as a strategic controller for the RTS, building and scheduling reactive behaviors with known resource requirements. Within the AIS, tactical methods are used to actually build the reactive plans: these methods may sacrifice the quality of the plan in response to resource limitations. The AIS essentially performs the same task as deliberation scheduling, deciding what tasks should be executed by the RTS at any time. Unlike deliberation scheduling, CIRCA's AIS does not require performance profiles and iterative tasks, and is capable of building useful guaranteed task schedules even when tasks have very complex interactions and dependencies. Deliberation scheduling is able to analytically derive an optimal schedule given performance profiles; CIRCA requires less precise information and performs a *search* for a desirable task schedule that yields acceptable output quality within given resource bounds.

CIRCA implements this search by iterating over a loop that first has the AIS plan a set of tasks to meet a given output quality threshold, and then runs the Scheduler on those tasks to see if they can all be successfully run given the system's limited resources. In essence, this process corresponds to choosing a point along the performance profile (for the overall system, not a single task) that is above the quality threshold, and then using the Scheduler to check if that point is also below the maximum resource usage threshold (e.g., point A in Figure 2.4). Failure to produce a schedule is an indication that the chosen set of tasks, while providing sufficient output quality, requires too many resources (e.g., point B in Figure 2.4). This iterative process of choosing a set of tasks to achieve a given level of output quality and then checking their resource usage with the Scheduler can be viewed as an any-quality algorithm: the iteration will continue until a feasible schedule of tasks is found that exceeds the desired quality threshold. Because this generate-and-test technique does not rely on explicit knowledge of the form of the performance profiles for each task, it is more widely applicable than the deliberation scheduling technique.

The RTS executes the reaction plans built by the AIS and Scheduler. In addition to the schedule of tasks that must be guaranteed to meet their deadlines, the AIS can also send the RTS a list of "if-time" or "best-effort" tasks, that should be run only if unused resources become available. Viewed from a high level, the RTS functions as a generalized form of imprecise computation: the guaranteed tasks represent mandatory computation, and the if-time tasks are optional computations. However, while the imprecise computation method specifies that optional computations are any-time algorithms that will improve the quality of the mandatory computations they follow, CIRCA's if-time tasks need not be incremental, and they may have little or no relation to the tasks they follow. If-time tasks are simply those tasks which the AIS/Scheduler decided were desirable, but which could not be fit into the schedule of guaranteed tasks.

Thus, CIRCA combines aspects of both strategic and tactical methods in addressing real-time intelligent control problems. In the next chapter, we provide a more detailed view of CIRCA, and compare the architecture with several closely-related systems.

CHAPTER 3

OVERVIEW OF CIRCA

We assume that the system CIRCA controls will inhabit an environment in which, to survive and achieve its goals, the system must respond actively to various types of inputs. 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 [73] and bounded reactivity [56] 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 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 test expression (or precondition), an action expression to evaluate if the test returns true, data about the sensing and actuating resources the TAP requires, and worst-case timing data on how long it takes to test the precondition and execute the action. During the process of building control plans (to be discussed in detail in Chapter 5), individual TAPs are automatically generated by composing primitive descriptions of actions and tests. The planning process also assigns

```
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 3.1: An example TAP from the Puma domain.

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 3.1 shows an example TAP generated automatically for the Puma 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).

To facilitate our discussion, we introduce a functional notation for referencing features of a TAP τ. The function $test(\tau)$ refers to the TAP's test expression, and $action(\tau)$ refers to the action the TAP implements. We use $wcet(test(\tau))$ to refer to the worst-case execution time of the TAP's test expression, and likewise $wcet(action(\tau))$ for the worst-case execution time of the TAP's action. 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 $wcet(\tau) = wcet(test(\tau)) + wcet(action(\tau))$. 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 Chapter 4; CIRCA does not represent or reason about them.

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 test expression of a guaranteed TAP in the schedule returns false, then an unguaranteed TAP may be executed in the time scheduled for that guaranteed TAP's action.

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 into and out of the RTS is encapsulated within TAPs, so that all RTS activity is scheduled explicitly (see Section 6.2). 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 are anticipated during and after the accomplishment of this goal. The effect of the latter criterion, which will be explained in detail in Chapter 4, is to allow the 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. Most of the work on CIRCA's AIS has focused on the planning processes that reason about a world model to build control plans; the model and planning methods are described in detail in Chapter 4 and Chapter 5.

TAP control plans can easily 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. The TAPs for each action are simply built with tests that are activated by the postconditions of previous TAPs in the sequence. 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[1].

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

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

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 the Puma 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 Puma 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 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 often 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, CIRCA 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[2]. 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 can make 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

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

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 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. In the following section, we explain why CIRCA's automatically guaranteed control performance is superior in many ways to unguaranteed control.

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 the Puma 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 [21]. 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 way to build a control system without such assumptions: 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 [76], 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.

3.5 Summary of the CIRCA Approach

The concepts and goals of CIRCA can be characterized in several useful ways, providing different viewpoints on the architecture. Each of these viewpoints is valid, but each stresses different aspects of the approach:

- **CIRCA as a real-time AI system.** CIRCA's goal is to be "intelligent about real-time," as opposed to being "intelligent in real-time." That is, CIRCA's AI processing is not constrained to meet deadlines. Instead, the RTS is responsible for executing reactions that are guaranteed to meet the domain's hard deadlines, while the AIS executes less-predictable search algorithms that address task-level problems without hard deadlines. The reactive plans executed by the RTS are built specifically to restrict the progression of world states so that failure is avoided and the state of the world remains within the range of the plan's applicability. In other words, the RTS will prevent failure and keep "stalling" the domain until a new plan is downloaded. Thus CIRCA is able to apply unrestricted AI methods to difficult task-level problems while also guaranteeing control-level responses that will meet deadlines.

- **CIRCA as an any-completeness method.** CIRCA meets the demands of real-time control within a bounded-reactivity system by guaranteeing that it will produce a precise, high confidence response in a timely fashion *to a limited set of inputs.* In other words, the architecture can sacrifice completeness of attention in order to achieve precision, confidence, and timeliness in its responses to environmental changes that it does observe.

- **CIRCA as an introspective system.** The key to CIRCA's performance guarantees is its ability to introspect on its own performance, recognizing its own resource limitations and the resource needs of the reactive TAP plans it is generating. The AIS can be viewed as reasoning about a fixed meta-level of the RTS; while the RTS executes the TAP schedule to decide what to do next, the AIS plans the next schedules for the RTS. In addition, the AIS itself has meta-level capabilities that allow it to introspect on its own deliberative behavior. For example, the AIS can recognize when it is taking a long time to generate plans, and may decide to simplify its planning process as a result.

- **CIRCA as an automated system designer.** Traditionally, real-time systems have been designed by humans, who are given detailed characterizations of how the real-time system needs to interact with the environment. CIRCA is a first attempt at automating the entire process of building a real-time system, from planning tasks, to deriving their constraints, to scheduling them, and finally to executing them predictably. By automating this entire design and implementation process, CIRCA is able to dynamically and flexibly develop and modify its real-time behavior in the face of changing goals, capabilities, and/or domains.

Figure 3.2 shows a flowchart mapping the steps of a traditional control system design process to related portions of the CIRCA approach. Beginning in the upper left of the figure, the designer (human or automated) is given a specification of the system to be controlled; in the case of CIRCA, this specification has three parts: a set of initial world states, a set of state transitions that describe how the world can change, and a set of agent capabilities, describing how the agent can change the world. The output specification describes the desired behavior; for CIRCA, the specification includes both goals of avoidance (to stay out of some undesirable situations) and goals of achievement (to attain some desirable situations). Chapter 4 describes the CIRCA world model in detail.

The design phase of the process builds a tentative control system; CIRCA builds a TAP control plan using the planning methods described in Chapter 5. The next phase of the design process is to verify that the proposed control system meets the specifications; CIRCA verifies the logical correctness of a control plan when it is built, based on the world model, and the Scheduler verifies that the plan can be executed successfully by the RTS, as described in Chapter 6.

Following the dashed arrows in the flowchart, it is also possible for the design or verification phase to fail, indicating that some modifications must be made to the initial design or the specifications. Such modifications are essential to automating the overall design process, for two reasons. First, because heuristics are used to generate designs, the initial proposed design may not actually meet the specifications. A mechanism must be available to modify the planning process (or some other system aspect) so that a different design is heuristically generated and tested. Second, because CIRCA is intended to control an autonomous agent with bounded resources, it is not possible to ensure that the agent will always have sufficient resources to accomplish every task that might arise. Essentially, there are too many possible combinations of input and output specifications to enumerate. As a result, CIRCA must dynamically consider how to apply its limited resources to best achieve its goals, possibly by preferring some goals over others, by changing plans, or by making other modifications to the planning process or specifications. This requirement distinguishes CIRCA's approach from a more traditional design process, in which the goal is only to design a system that meets the fixed input and output specifications. In contrast, CIRCA may actually have to modify the I/O specifications of its control system design, when faced with resource limitations.

CIRCA has two ways of recognizing when modifications are necessary: the planner may fail to produce a plan, or the Scheduler may be unable to build a feasible schedule. In response to these conditions, CIRCA can make modifications to an individual control plan, or to the state transition or goal specifications used to derive control plans. These modifications allow the system to make performance tradeoffs to account for overconstraining domains, and are described in detail in Chapter 7.

3.6 Comparison to Related Work

Having presented a general description of CIRCA's approach to real-time AI, we are now in a position to compare this approach with several closely-related agent architectures. We will focus largely on the architectural division of responsibilities in these systems, as well as the types of performance guarantees they can provide, and thus how well they address real-time control issues. There are three main approaches to developing intelligent real-time control systems: embedding an AI system within a real-time system, embedding real-time reactive elements within an AI system, and using cooperating reactive and deliberative systems.

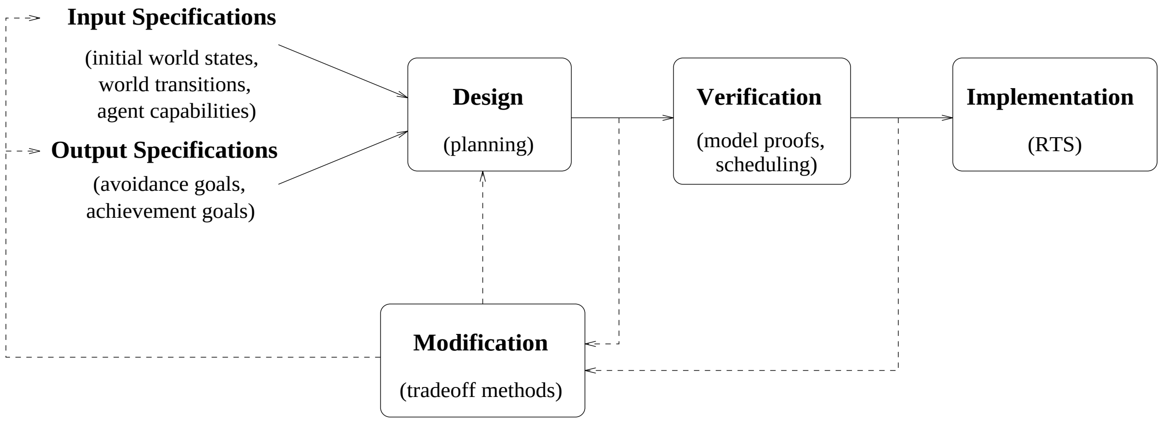


Figure 3.2: A flowchart showing the stages of real-time system design.

3.6.1 Embedding Intelligence in a Real-Time System

The most straightforward approach to real-time AI is to embed intelligence within a real-time system, so that the AI mechanisms are required to meet deadlines— the goal is to be "intelligent in real time." One way to accomplish this is to simplify an AI system's knowledge-base and inference mechanism so that it responds to all inputs within a bounded time [36, 40]. Unfortunately, this approach engineers out of the AI system the high-variance search and unpredictability which distinguishes AI techniques from simple algorithms. In a sense, when a system with these limitations can always solve a problem, that problem is no longer in the realm of AI.

Several other, less-restrictive approaches have been used to embed AI methods within real-time systems. These approaches variously rely on real-time operating systems, constant-cycle-time circuits, or any-time algorithms to enforce guaranteed, predictable execution.

CROPS5

CROPS5 is a C-based parallel implementation of the OPS5 production system [60]. 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.

Subsumption

The subsumption architecture [5] consists of numerous small finite state machines ("modules") running in parallel with no shared memory, connected by simple message-passing channels. The modules are "reactive" in that they maintain little or no internal state, and rapidly produce outputs in direct response to inputs, with a minimum of inferencing. These systems do not perform the lookahead (or internal simulation of actions) that classical planning implies; instead, they essentially act as a parallel set of situated-action rules that recognize an input and produce the associated output immediately. Subsumption systems incorporate hierarchical control by having higher level modules alter the input or output of lower level modules. By inhibiting or enabling the output of different modules, a high-level module can activate a subset of desired behaviors in much the same way as CIRCA chooses reactions to guarantee.

The processor-per-behavior approach assumed by subsumption suffers from obvious scaling problems, and also wastes computing power, since many behavior processors may not be active at all times. Our approach is based on the assumption that, as we extend the range of situations that our system is prepared to encounter, the necessary behaviors will become too numerous to allocate each to a separate processor. CIRCA's Scheduler module addresses this problem by scheduling TAPs for the single-processor RTS.

Other systems [8, 75] have made provisions to activate only subsets of reactive behaviors implemented on a single processor. However, these systems do not reason about the resources required for each set of behaviors, and do not use advanced AI techniques to control the set of activated behaviors[3]. CIRCA's resource allocation and scheduling are crucial to the system's flexibility, extensibility, and efficiency. Furthermore, by explicitly reasoning about time and resources, CIRCA is able to provide guaranteed performance, which reactive systems cannot. Reactive systems simply run as fast as they can, and thus they are only "coincidently real-time" [36].

Finally, since purely reactive systems lack the ability to learn and to form complex symbolic plans or expectations, they have little of the power we associate with intelligent systems [26]. Essentially, all of the inferencing and uncertainty associated with intelligent behavior has been engineered out of these systems. We might consider them to be convenient, powerful formulations of traditional control systems, rather than intelligent real-time control systems.

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 [63, 64]. 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.

[3] Although Connell and Viola [8] are on a similar track: they use a human to make the decisions.

Thus Rex machines provide predictable execution and support the types of performance guarantees enforced by CIRCA's RTS.

Gapps [32, 33] 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.

Any-Time Algorithms

Any-time algorithms (as discussed in Chapter 2) have recently become popular in both the AI and real-time communities. Some high-variance AI methods can be cast as any-time algorithms, which are then able to make timeliness guarantees because they can be interrupted at any time. However, the quality or correctness of the result returned after a deadline-driven interrupt cannot be guaranteed. Thus any-time algorithms may sacrifice precision, completeness, or other quality measures for timeliness, while CIRCA strives to guarantee both quality and timeliness. 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 [41] 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. CIRCA uses this technique when generating TAP plans (see Section 5.3.1), where a minimally acceptable plan achieves only the control-level goals.

PRS

CIRCA's AIS includes some mechanisms derived from the Procedural Reasoning System (PRS) [23, 31], which itself has 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 PRS' inferencing capabilities and make guarantees about overall response time [31]. This approach leads to a complete embedding of the AI system within the real-time application environment [57], and requires either low utilization or engineering out the high-variance AI processing.

The guarantees that PRS makes are external to the system's operation: it does not introspect on its abilities. PRS also does not plan in the sense of reasoning about an environment and the appropriate actions; instead, it chooses how and when to invoke procedural "knowledge areas," which are themselves partial plans.

3.6.2 Embedding Reactivity in an AI System

Other research projects have taken the opposite approach, embedding real-time capabilities within an AI system. These systems use a set of designated reactions which bypass the normal invocation mechanisms, leading to faster response times.

For example, the Soar system [39] is an enhanced production system that structures all deliberate activity as search. Searches are conducted in problem spaces characterized by current states, goal states, and operators to move between states. Soar productions encode knowledge about what decisions to make in different situations. To make a decision (choosing what goal to pursue, operator to apply, etc.), Soar tries to match and fire all of its productions repeatedly, until no new productions match. The decision is then made based on all the knowledge retrieved from the production firings. If the productions do not provide enough knowledge to make a decision, the system recursively subgoals to solve the problem of "making the decision." The integrated Soar learning mechanism ("chunking") builds new productions that summarize the search performed to solve problems.

From a predictability perspective, Soar's flexible decision-making approach has the disadvantage that arbitrarily large amounts of subgoaling and production matching may occur. To avoid subgoaling, Soar encodes one type of reactive knowledge as productions that indicate particular operators *must* be selected in a given situation [37]. However, this reaction technique still incorporates the uncertain delay associated with firing all productions until quiescence and then making the decision to implement the chosen operator. Recent work by Doorenbos [12] has shown significant performance improvements for Soar's matching phase with very large numbers of rules ($> 100,000$), but the match time can still rise as the number of productions increases. Even if the match time was a known constant, the process of firing productions repeatedly until quiescence is still an uncertain computation subject to scaling with the size of the knowledge base.

An even faster but less-controlled form of reactive knowledge can be implemented by productions that directly create motor commands when they are matched and fired, independent of the post-quiescence decision mechanism. This approach eliminates much of the potential for the interference of unpredictable search, but also moves the reactions below Soar's level of introspection— the system cannot inspect or modify its productions directly, and thus such reactions would be outside of its direct control.

CIRCA addresses the issues of matching cost and iteration by choosing the subsets of reactive knowledge it will test during each cycle of the RTS. These choices prevent CIRCA from displaying the completely opportunistic behavior of general pattern-directed invocation methods, but they are necessary to cope with restricted resources and bounded reactivity. The choice of TAPs also has the effect of focusing the system's attention on features which are deemed important, eliminating the assumption that all changes in the world are detected by the sensor system [38]. By planning and reasoning about sequences of its own reactions, CIRCA can provide guarantees on its overall interactions with the environment, in addition to individual reactive behaviors.

One significant advantage of the Soar approach is that the benefits of automatic learning (a major focus of Soar research) are shared by both the deliberative and reactive processing. When Soar solves a problem, it chunks the result in a new production so that, in the future, the result will be immediately available. CIRCA's planning operations and the construction of reactive control plans might be viewed as the chunking of deliberation into reactive form, although that form is not a unified representation of knowledge as in Soar. Currently, CIRCA does not store these "learned" reactions beyond their use by the RTS, but a case-based approach to plan retrieval could certainly be integrated easily into CIRCA's AIS.

3.6.3 Cooperative Systems

CIRCA demonstrates an alternative to the embedded approaches, using separate, concurrent AI and real-time subsystems to cooperatively produce the desired performance. Several recent projects have taken similar approaches, with a variety of different areas of focus.

ERE/RAPs

Hanks and Firby [26] are combining a transformational planner [27] with an execution module based on Reactive-Action Packages (RAPs) [15]. Each RAP is a separate entity that pursues a goal, possibly with multiple methods, until that goal is achieved. In pursuing a goal, RAPs can process global world model data and execute actions that change the model and/or the outside world. RAPs can also place new RAPs on the execution queue and suspend themselves, implementing sequential and hierarchical control. The description of the combined system [26] notes that sensing actions must be explicitly included within RAPs, so that data examined by the RAPs is up-to-date. CIRCA's TAPs make this even more clear: sensor data is acquired by individual TAPs, and the fact that sensor data becomes outdated is explicitly represented by the TAP frequency requirements stating how often the sensing TAP must run.

The RAP interpreter, running on a single computer, acts as a dynamic, non-preemptive multiprocessing scheduler, choosing the next RAP to run from a queue. This is significantly different from CIRCA's approach, in which the Scheduler builds a static schedule off-line from the execution system. Since RAPs are non-interruptible and their hierarchical computational complexity is not restricted, the RAP-based control system is not able to provide timeliness guarantees. Also, the strategic planning and RAP execution subsystems share a global world model; this shared resource could lead to contention problems that would unpredictably delay the subsystems. CIRCA avoids shared data for this reason, relying instead on message passing and interrupts.

Hanks and Firby note that the RAP structure provides a useful representation which can be used by the planner for reasoning about execution, and, without translation, by the execution system for the actual control of operations. CIRCA's TAPs provide exactly the same shared representation. However, Hanks and Firby focus on meta-control problems like deciding when to continue lookahead planning and when to interrupt a current plan to install a new one. We focus instead on establishing the predictable mechanisms which will allow such policy decisions to be rigidly enforced.

AuRA

Arkin's Autonomous Robot Architecture (AuRA) [4] includes a reactive execution subsystem and a hierarchical planner that determines which reactive "schemas" are active. A world modeling subsystem controls AuRA's stored knowledge, providing an interface that avoids shared-memory assumptions. AuRA's reactive schemas are essentially formulas for calculating vector fields that describe the robot's desired motion for a particular behavior. For example, an obstacle-avoidance schema outputs a navigation vector moving the robot away from the obstacle. The vectors from different active schema are combined via vector summation and normalization. This technique is an elegant method of "command fusion," the combining of simultaneous control commands from multiple sources. CIRCA does not address command fusion directly; in fact, since TAPs are executed sequentially, there is never an opportunity to combine commands. However, the conditions that determine which TAPs fire may be seen as preempting command fusion, choosing instead a single TAP to implement the desired combinations of commands. While CIRCA's method is less intuitive in some cases, AuRA's vector fusion is overly simplistic, because it may not always be desirable to merely sum commands. Sometimes one command should completely override another, and magnitude may not be a sufficient expression of that priority. Or, the confluence of two conditions triggering two schemas might warrant a response that does not resemble the sum of the individual responses. For example, cooking on a stove might prompt a response "stay near the stove," while a fire would trigger "move away from the fire." How can the magnitudes of those responses be arranged to coordinate with the overall desirable response to a stove fire, that might be "move closer to the stove to turn off the gas." For problems beyond simple numerical navigation, vector field formulas and vector summation are not sufficient reactive mechanisms.

AuRA also includes a "homeostatic control" subsystem that monitors the internal conditions of the execution subsystem, allowing changes in the execution subsystem to affect the planning process. CIRCA can provide similar fault-tolerant functionality, as will be discussed in Section 6.3.5. AuRA does not address the timeliness or resource restrictions that are the focus of our architecture.

TCA

Simmons' Task Control Architecture (TCA) also combines reactive and planning systems [70, 72]. The architecture itself provides for a central control module, a set of distributed task-specific processing modules, message-passing between modules, and a task representation ("task trees") that coordinates planning and execution. The central control module maintains the task trees that represent the system's plans, and issues messages to task modules as the task trees are traversed. In response to these messages, task modules may implement task-specific planning operations, sensing strategies, or motor control. Constraints among task tree branches can restrict the central module's processing of the tree, making the system wait for completion of one task module operation before initiating the next. Task trees may also include polling monitors that periodically check to make sure some condition is true in the world (by querying a task module), as well as interrupt-driven monitors by which task modules can alert the central control module.

TCA thus provides the ability to overlap or interleave distributed planning and execution, and its monitors yield some reactive capabilities. However, the central control module represents a severe bottleneck through which all messages must pass. For example, there is no direct pathway between a sensing module and a motor control module. Since the central control module can become involved in updating arbitrarily large task trees, its performance is uncertain, and TCA cannot provide the timeliness guarantees required for hard real-time control tasks.

Although TCA does not provide execution-time guarantees, it does reason about its limited sensor capabilities, and is intended to derive sensing parameters (such as frequency) from a causal explanation of the sensing behavior and environment. This corresponds directly to CIRCA's reasoning about TAP parameters. However, although sensing monitors are under the control of a central AI system, the reactive elements of TCA which attempt to keep the system safe are outside the system's control [71]. In contrast, CIRCA reasons explicitly about its ability to remain safe by activating selected sets of reactions, and thus CIRCA can take into account its own bounded reactivity in building plans and choosing courses of action.

ATLANTIS

Miller and Gat have developed the three-layer ATLANTIS system [53], in which the bottom layer provides a subsumption-like reactive controller and the top layer is a deliberative planner and world modeller. In between, the sequencing layer turns on and off sets of reactive behaviors, much as CIRCA runs different TAP schedules. The sequencing layer actually does more, since it also maintains a task queue similar to the RAP interpreter, and sequences these tasks when it is interrupted or detects that the previous task is finished. ATLANTIS does not address the resource reasoning or guaranteed performance objectives of CIRCA.

DR/MARUTI

Hendler and Agrawala [30] 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 [29]. 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 [30], 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.

Universal Plans

Schoppers' research on the automatic generation of Universal Plans (UPs) [66, 67] 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 maintain the system's safety while also making progress towards its task-level goals. Schoppers [69] has recently discussed how UPs can similarly keep a system safe through 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.

$\mathcal{RS}$

Lyons *et al.* [45, 46] 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 CIRCA represents both. Rewrite rules describe the evolution of $\mathcal{RS}$ systems, and these rules can be used to derive proofs that systems will meet their goals [46].

$\mathcal{RS}$ 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 [45]. A planning technique has also been proposed [45], in which a concurrent planning process incrementally modifies the reactive schemas running on the execution system. A major advantage of this approach is that it avoids CIRCA's behavior of building plans from scratch following every change of goal or other environmental feature.

RPL/XFRM

While $\mathcal{RS}$ uses a process-based representation for plans, McDermott is investigating a very general Lisp-like Reactive Plan Language (RPL) [49, 50] as a basis for both planning and execution. The XFRM system [51] includes a planner that incrementally modifies an RPL program to improve its performance on given tasks.

On the positive side, the flexibility of RPL gives tremendous representational power. The converse, of course, is that any planner that can automatically modify such code must include complex mechanisms for reasoning about program structures such as variables, loops, and conditionals. Furthermore, XFRM does not generate RPL plans given goals and a description of the environment; instead, it modifies pre-built plans provided by the system designer, that are assumed to already have some reasonable level of competence. McDermott has noted that it is very difficult to encode a new domain in XFRM because of the large amount of *a priori* plan knowledge which must be given to the system, and also because of the difficulty of hand-coding plan critics that can recognize and fix problems with RPL code. The RPL representation is so flexible and powerful that simply reasoning about the meaning of a program is an AI task unto itself.

3.6.4 Comparison Summary

Most of the systems described above are Turing complete— they can each implement almost any functionality we can specify. Thus the real issue in comparing these systems is not their absolute computational capacity, but how "naturally" each system's capabilities match with a specific type of problem. CIRCA has been specifically tailored for the demands of real-time intelligent control domains, while many of these related systems were designed for different problems. Thus it is not surprising that CIRCA provides a unique combination of features, as shown in Table 3.1. The features relevant to our concern with real-time intelligent control include:

Predictable reaction time: Is there a firm bound on the time that the system will require to begin reacting to an event? If not, the system is not suited to hard real-time domains.

Predictable response time: Is there a firm bound on the time that the system will require to finish reacting to an event? If not, the system is not suited to hard real-time domains.

Introspection and guarantees: Can the system reason about its own capabilities to make guarantees on its own performance? If not, the system lacks the power to recognize overconstraining situations where performance tradeoffs may be necessary.

Planned performance tradeoffs: Can the system make decisions about trading off the goals it will pursue? This flexibility is required in overconstrained domains where all the initial goals may not be possible.

Parallel deliberation and reaction: Can the system react to the environment while also deliberating about future behaviors? If not, the system may become bogged down in highly dynamic environments where it must use all of its resources on short-term reactions.

Unrestricted search-based planning: Is the system capable of performing the high-variance search-based deliberation tasks characteristic of AI?

Incremental plan modification: Can the system incrementally improve its plans, rather than rebuilding them from scratch? This approach may have efficiency advantages in some situations, particularly if integrated with case-based approaches.

Metric time model: Does the system include a metric model of time? If not, the system may not be able to reason about hard deadlines.

Selective Perception: Does the system choose which environmental features to sense, as opposed to assuming that a complete world description is continuously available? This functionality is useful when sensing activities are costly or otherwise constrained.

Learning: Does the system improve its performance over time? While systems that learn from previous (possibly incorrect) performance are not suited to hard real-time domains, it is certainly true that learning in general is a useful capability, particularly for resource-constrained systems.

Feature	CIRCA	PRS	UPs	$\mathcal{RS}$	Soar	ATLANTIS	GAPPS	XFRM	Subsumption	DR/MARUTI
Predictable reaction time	$\surd$	$\surd$	$\surd$			$\surd$	$\surd$		$\surd$	$\surd$
Predictable response time	$\surd$	$\surd$								
Introspection and guarantees	$\surd$									
Planned performance tradeoffs	$\surd$									
Parallel deliberation and reaction	$\surd$		$\surd$	$\surd$	$\surd$	$\surd$		$\surd$		$\surd$
Unrestricted search based planning	$\surd$	$\surd$	$\surd$		$\surd$	$\surd$		$\surd$		$\surd$
Incremental plan modification			$\surd$	$\surd$	$\surd$			$\surd$		
Metric time model	$\surd$									
Selective perception	$\surd$	$\surd$	$\surd$		$\surd$	$\surd$				
Learning					$\surd$					

Table 3.1: Summary chart comparing system capabilities. $\surd$ indicates that the system has demonstrated this feature.

CHAPTER 4

THE WORLD MODEL

This chapter describes in detail the world model that CIRCA uses to build reactive TAP plans. The model allows the system to represent and reason about both the dynamic environment and the system's own actions. We begin by describing the model informally, and then provide a more formal notation to add precision to our discussion. Using this formalism, we describe the conditions under which portions of the world model can be considered "safe." Planning safety-preserving reactions thus becomes a matter of finding actions that can be proven to keep the system in such regions of the model space. After detailing the proofs involved in showing that a particular plan can keep a system safe, we show how CIRCA can safely string together sequences of these control-level plans to achieve longer-term task-level goals.

We then discuss several unusual aspects of CIRCA's world model which result from the simplifications made to accommodate real-time considerations. This chapter concludes with a brief evaluation of the model's representational power and a final summary of the information that must be captured in the world model to make real-time guarantees possible. The planning mechanisms that use the world model are described in Chapter 5.

4.1 The Informal View

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"– that is, the status of a process is considered a feature of the world (a "fluent" [48]), 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 free) (robot-position over-box)**.

4.2 The Formal View

We now describe a more formal representation of the world model which will be useful for 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; $T_i : S \rightarrow S$. The functions $D : T \rightarrow S$ and $R : T \rightarrow S$ determine the domain and range of a transition; $T_i : D(T_i) \rightarrow R(T_i)$.

Figure 4.1 shows an abstracted representation of a small portion of the graph model for the Puma 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}. To obtain a reasonably compact example, many states and transitions have been omitted from this diagram, and even the state descriptions include only 7 of the 11 features used in the actual domain model. State $\mathcal{A}$ in the figure represents the world state in which the robot is idle, no parts have yet appeared, and there is no emergency alert. The solid single arrow from state $\mathcal{A}$ to state $\mathcal{B}$ represents the event transition indicating that a new part has arrived on the conveyor. The dashed arrow from state $\mathcal{B}$ to state $\mathcal{C}$ represents the start of a planned sequence of action transitions that have been planned to pick up the part and place it in the box. Within that sequence, state $\mathcal{E}$ represents the world state in which the robot has picked up the part from the conveyor and is moving to place it into the box. The double arrow to state $\mathcal{G}$ 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 $\mathcal{H}$.

The solid single arrow from state $\mathcal{E}$ to state $\mathcal{J}$ represents the possibility that the emergency light may go on while the robot is in motion[1]. From state $\mathcal{J}$, the double arrow to state $\mathcal{F}$ (failure) represents the deadline for reacting to the emergency and pushing the button. The dashed arrows to states $\mathcal{K}$, $\mathcal{L}$, and $\mathcal{M}$ 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 (**stop-moving**, **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.

4.3 State 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 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 4.1. 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 4.1, the $min\Delta$ for the temporal transition from state $\mathcal{E}$ to state $\mathcal{G}$ 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 the robot could ever arrive over the box (and

[1]We have omitted other instances of the same event that may occur, for example, from state $\mathcal{B}$ and state $\mathcal{C}$.

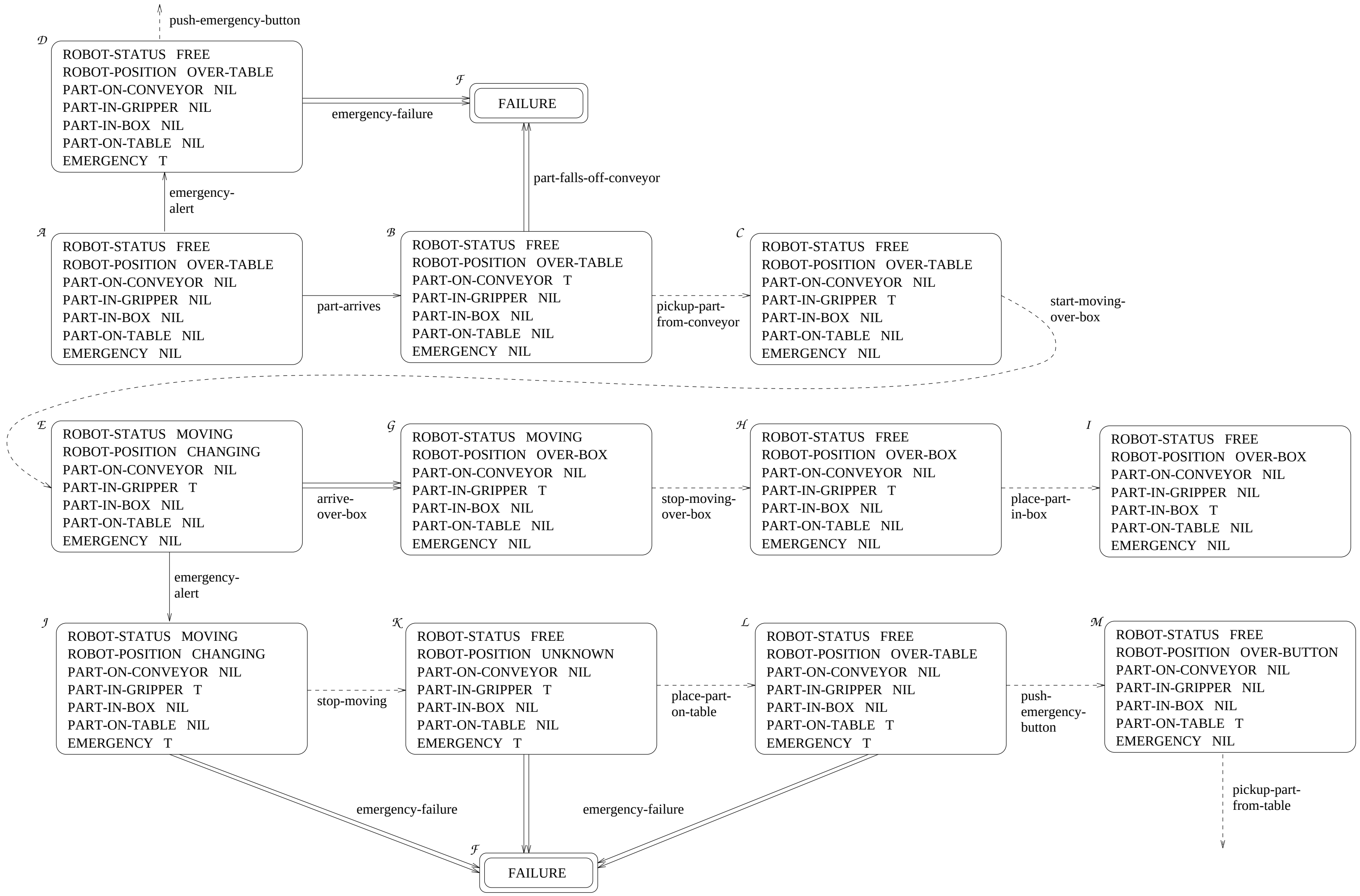


Figure 4.1: An abstracted portion of the world model for the Puma domain. For clarity, many states, state features, and transitions have been omitted.

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

Table 4.1: Enabled interval definitions.

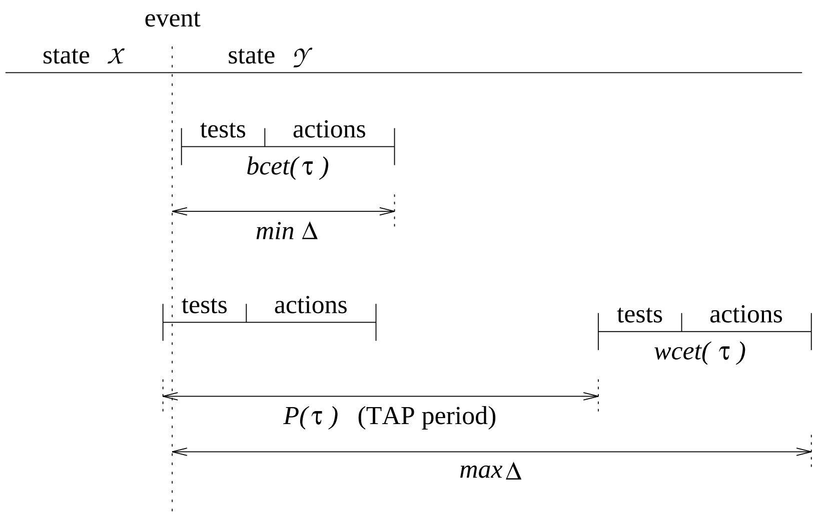


Figure 4.2: Deriving the $min\Delta$ and $max\Delta$ for an action transition implemented by a periodic TAP.

thus enter state $\mathcal{G}$).

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 4.2. 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.

4.4 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. In the example of Figure 4.1, the entire graphed set of states $\{\mathcal{A} - -M\}$ is event-closed. Note that, in the complete Puma world model, this set is not event-closed because the **emergency-alert** event transition might lead out from some of the states where it is not shown here. For the purposes of this discussion, we will consider only the states and transitions shown in the figure.

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*. Looking again at the example in Figure 4.1, the entire graphed set of states is also safe, because the only transitions to the failure state $\mathcal{F}$ are temporal transitions.

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 [69] which is known to restrict the operation of a controlled system to a desirable range of states. In our running example, if we could show that the actions **stop-moving**, **place-part-on-table**, and **push-emergency-button** all are known to occur before the respective temporal transitions to failure, then the failure states could be removed from Figure 4.1, and that set of world model states would be safely-controlled.

To show formally 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 [18], 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.

4.5 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 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: $R_I = \bigcup\limits_{Ii \in I} R_{Ii}$.

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 4.2 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

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

Table 4.2: Conditions for removing world model states and transitions.

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 steps and tries to build complete control plans for each step. 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.

It should be noted that the model transformation method described above is not a very practical method of automatically deriving complete control plans, because it involves enumerating the entire state space of the world model, and then progressively pruning out undesirable regions of that state space with planned actions. The enumeration required at the start of this process is impractical in any reasonably complex domain, because of the combinatoric explosion that results from trying to describe the entire world in each world model state. For example, the eleven-feature description we typically use for the Puma domain has 5120 possible combinations of feature values. Instead of building a representation for each of these combinations and then pruning many out, we instead use a forward-chaining method to simultaneously derive control plans and build up the representations of the reachable world states. The implementation details of this process are described in Chapter 5.

4.6 Transitions between Safely-Controlled Sets

The conceptual goal in restricting the world to a safely-controlled set of states is that, while the RTS executes the reactive TAPs that keep the world in that set of states, the AIS will work on building a TAP plan for the next useful set of model states. When the new plan is prepared, it is downloaded to the RTS and the world can progress into the new safely-controlled set of states.

To ensure the safety of the system even during this transfer between safely-controlled

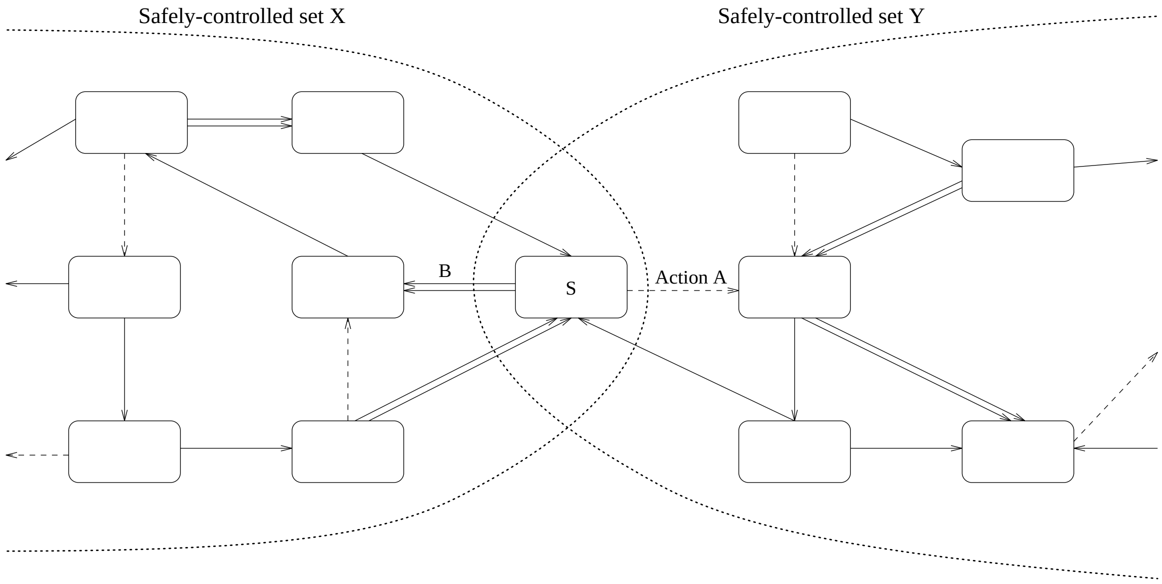


Figure 4.3: The state overlap required for a transfer between safely-controlled sets of states.

sets of states, the sets must satisfy a simple criterion: they must overlap, as illustrated in Figure 4.3. In this case, the two sets **X** and **Y** share the state **S**. The RTS is initially given a TAP plan that keeps the world restricted to the **X** set of states, on the left. When a new plan is available for the **Y** set of states, *and* the world is in state **S**, then the RTS can transfer control to the new TAP plan, which will execute the action **A** to progress into the new set of states.

Chapter 6 includes additional details on exactly how this transfer of control is accomplished in our prototype implementation. From the modeling perspective, however, an important point to note is that, because the transfer state[2] is shared by the two sets, both of the corresponding TAP plans are capable of handling the world and avoiding failure from that state. Thus when the TAP plan for set **Y** is first executed on the RTS, it is has been planned to cope with the current state **S**.

There cannot be any event transitions from the shared state to a non-shared state, because that would prevent the system from being sure that the transfer of control to the new TAP plan is accomplished in a state for which the new TAP plan is prepared. Temporal transitions from the shared state to a non-shared state are acceptable if they return to the first set of states (as illustrated by the transition **B** in Figure 4.3). However, if such temporal transitions exist, CIRCA must ensure that they are preempted by an action transition in the new TAP plan. That is, the action **A** must preempt the temporal transition **B** to ensure that, once control is transferred to the new plan, the world does not slip back into the previous set of states.

[2]The overlap and transfer may occur in multiple world states; for simplicity, we have illustrated only a single overlap state.

4.7 Choosing Safely-Controlled Sets

Resource limitations are the primary motivation for dividing a long-term goal into a series of shorter-term plan steps: if a system has sufficient resources to continually observe, reason about, and react to all of the possible world situations it will ever encounter without missing any deadlines, then focused attention is not necessary. However, most realistic domains do not provide sufficiently vast resources, particularly if the domain requires high-variance AI methods. CIRCA is designed to deal with resource-limited domains by restricting its reactive attention to the states reachable within a safely-controlled subset. Therefore, the choice of exactly how the overall world state space is divided into safely-controlled sets is very important.

One simple approach to the problem of partitioning the overall state space into useful safely-controlled subsets is to incrementally decrease the size of the subsets, as necessary. The system could begin by trying to guarantee the entire set of goals over the entire state space. If such a plan is possible, resources are not a problem. More likely, the "universal" plan [66] will not be feasible, and the system will have to decide how to decompose the overall space, defining a useful set of intermediate goals that a TAP plan can achieve, and then pass control on to the next TAP plan. This iterative decomposition can continue, trying to build TAP plans for smaller and smaller sets of subgoals, until finally an acceptable decomposition is found.

The choice of exactly where in the state space to make the transfers between TAP plans is partially constrained by the conditions described above in Section 4.6. We have not yet developed stronger guidelines for this choice. It seems clear, however, that the partitioning of the state space into safely-controlled sets must be guided by heuristics. One promising idea is to define partitions between safely-controlled sets by finding planned actions that lead into time-constrained situations.

For example, consider a simple domain in which a mobile robot, equipped with an arm, is moving through an office to pick up an object from a table. The robot must avoid collisions with the walls and other obstacles. At a very coarse level of abstraction, the domain might be represented by the world model sketched in Figure 4.4. An initial plan might have the robot move towards its destination and, at the same time, move its arm into position to grasp the desired object. However, when the arm is extended to grasp the object, the effective radius of the robot is increased, so that it may collide with obstacles that are farther away from the robot body and sensors. If we consider that the obstacle detection routines have some limited range at which they detect looming obstacles, it is clear that the robot will have less time to react to avoid collisions when its arm is extended. Thus, the temporal transition to failure from state $\mathcal{E}$ has a shorter $min\Delta$ than the transition from state $\mathcal{D}$, where the arm is retracted. If we suppose that the system does not have sufficient resources to guarantee the higher reaction rate to avoid collisions from state $\mathcal{E}$, then this particular plan (and division of the space into subsets of states) is not feasible.

However, the planning system may recognize this problem, seeing that the action of extending the arm leads to the excessively time-constrained state $\mathcal{E}$. A simple solution is to postpone the action of extending the arm until after the robot reaches its destination, so that the robot's effective radius remains as small as possible while it is moving. The state

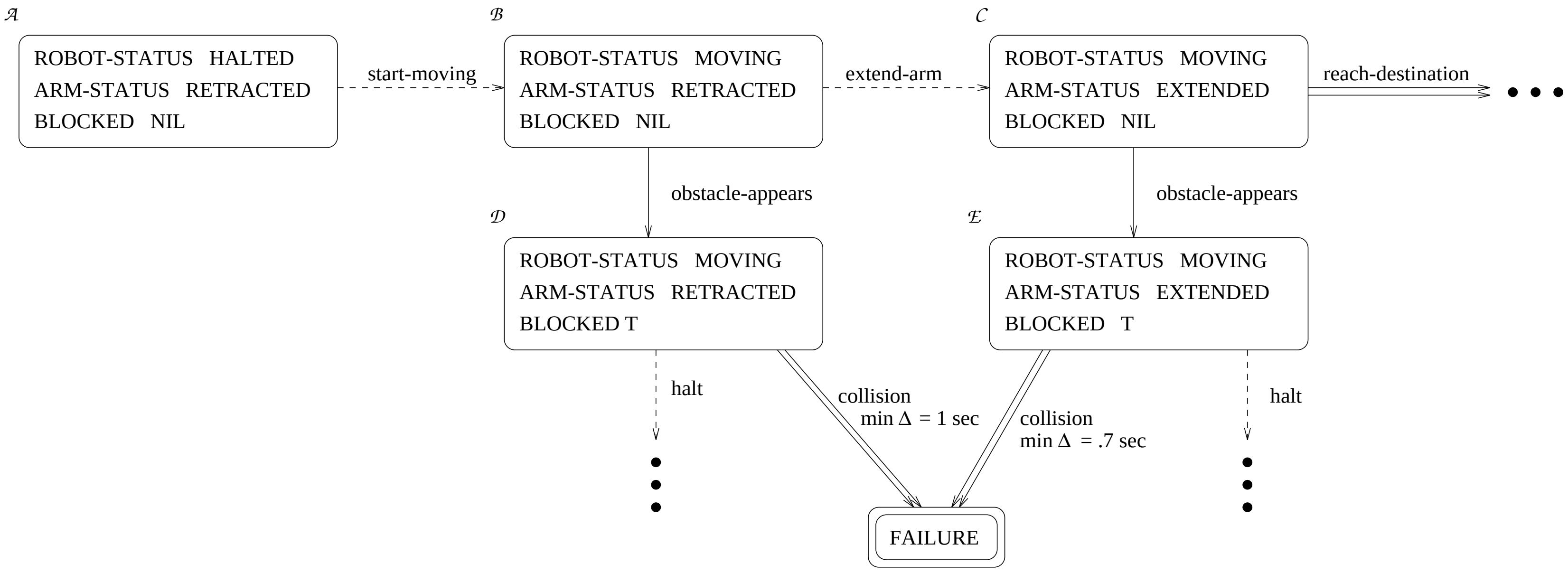


Figure 4.4: A simple mobile robot domain world model.

space will be transformed to separate the resource demands associated with moving the robot from the resources needed when the arm is extended. Then, if necessary, these two parts of the action plan (moving and reaching) can be separated into different TAP plans, with the transition between them occurring after the robot has moved to its destination. This approach to transforming the state space into safely-controlled subsets joined by an action corresponds to the intuitive notion of sequencing behaviors that cannot be safely performed at the same time. The key to this transformation is using the world model to recognize that the **extend-arm** action leads to states that cannot be handled, and thus that changes to that portion of the plan will be useful.

4.8 Relationship to Petri-Net Models

It is useful to compare this type of state-based model with models based on Petri Nets and their variations [61]. In Petri Net 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 Petri Nets, 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 Petri Net 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 Petri Net model.

4.9 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 5.3.2), nor does it mean that the control system must be able to perfectly track the progression of states in the environment [63]. 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 easily specify multiple range states, as shown in the example **pickup-part-from-table** transition in Figure 4.5. Here, the results of picking up a part off the table have been abstracted to show that, after the action, the table may still have a (different) part on it, or it may have no more parts on it. This nondeterminism helps the system avoid the state enumeration that would be associated with counting the actual number of parts that might have been queued on the table. This type of abstraction will be discussed in more detail in Section 5.4.5. The important point to note here is that the nondeterminism does not add any new sort of complexity to the world model; because all possible states must be handled, it matters little how those possible states are reached.

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 [3, 10, 27, 48, 77]. So far the only timing information we have shown for our world model is the simple worst-case values needed to recognize preempted

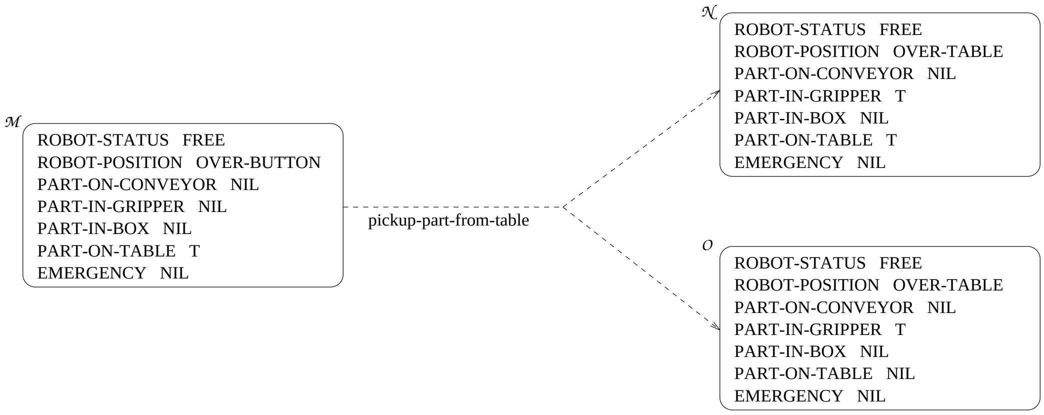


Figure 4.5: An example nondeterministic action transition.

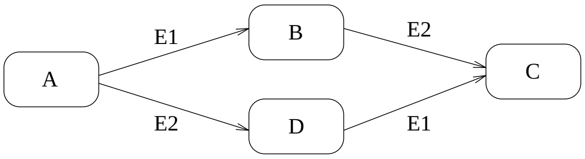


Figure 4.6: A world model subset showing the representation of potentially simultaneous events.

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 sequences of states that can occur without the passage of time. For example, in Figure 4.6 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.

4.10 Dependent Temporal Transitions

The world model has one difficulty with its minimalist representation of time: dependencies between temporal transitions. To illustrate the problem, Figure 4.7 repeats a portion of the Puma domain shown earlier. Beginning with the event **emergency-alert** entering state $\mathcal{J}$, the robot has ten 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 **stop-moving** and **place-part-on-table** does not remove the threat of failure from the emergency condition. Thus state $\mathcal{K}$ and state $\mathcal{L}$ still have temporal transitions to failure. The difficulty is that the minimum time until failure along these transitions is no longer ten seconds, because the emergency began in state $\mathcal{J}$, and some amount of time passed before we halted and moved to state $\mathcal{K}$. Thus the real minimum time to failure from state $\mathcal{K}$ depends on the sojourn time in state $\mathcal{J}$. 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 $\mathcal{K}$ has $min\Delta = 10$ minus the worst-case execution time of the TAP implementing the action from state $\mathcal{J}$ to state $\mathcal{K}$.

4.11 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 4.8 shows an action loop that the system might propose while building a plan for the Puma problem. In the figure, the system has planned to halt in state $\mathcal{D}$, transitioning to state $\mathcal{E}$. But it has also planned that, once in state $\mathcal{E}$, 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 $\mathcal{D}$, the time remaining until failure is not the original ten 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 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.

4.12 Predictive Sufficiency

Figure 4.9 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 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 4.9, 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.

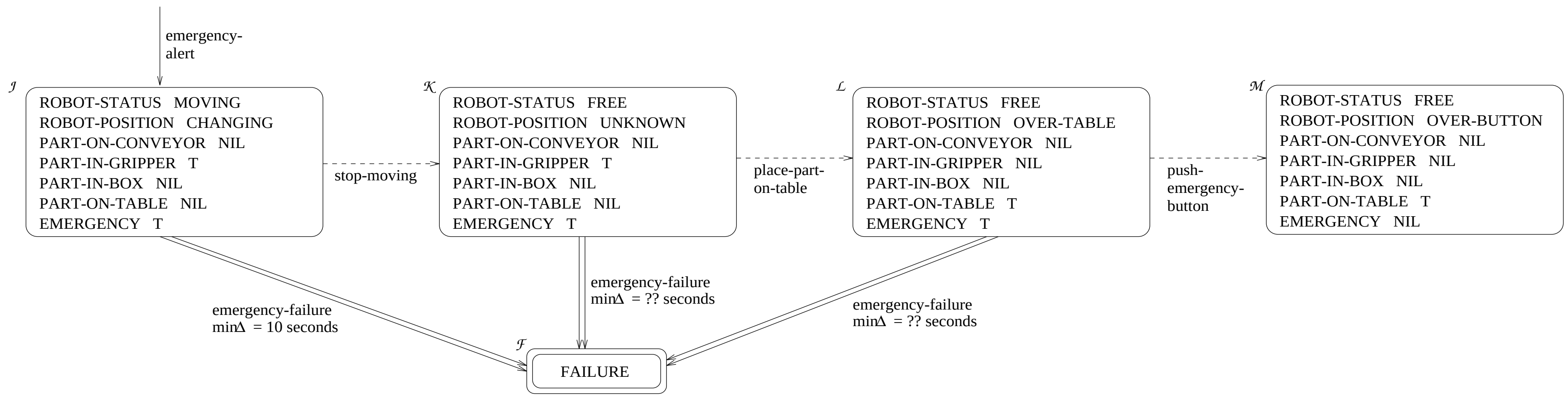


Figure 4.7: A portion of the Puma domain illustrating dependent temporal transitions.

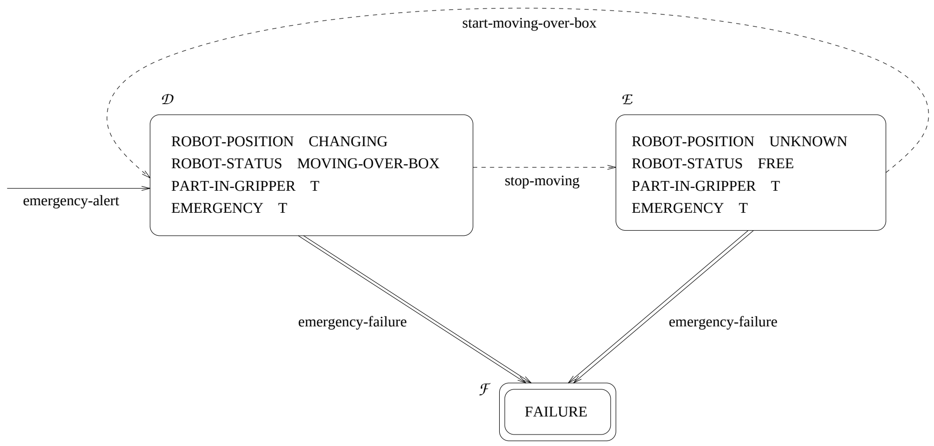


Figure 4.8: An action loop that might be generated for the Puma domain.

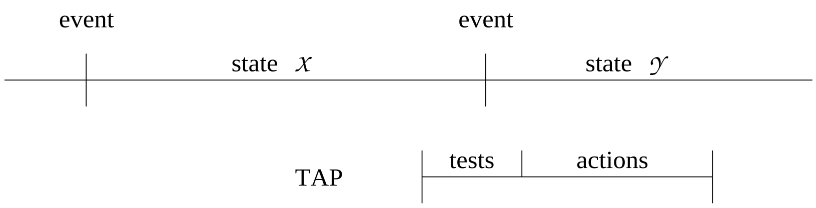


Figure 4.9: An inappropriate action.

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.

Formalizing and implementing techniques by which CIRCA can reason explicitly about the need for predictive sufficiency is one area of ongoing research; Appendix D presents more details on this topic. Currently, CIRCA does not explicitly reason about predictive sufficiency, and thus it cannot detect or plan to prevent potentially inappropriate actions. The system designer is responsible for ensuring that predictive sufficiency holds when necessary to avoid such problems.

4.13 Representational Power

Although many aspects of CIRCA's world model representation are fairly common, these aspects have been carefully combined to include precisely the information required for building real-time control plans. The representation of world states by simple lists of feature/value pairs is a powerful method essentially equivalent to first-order predicate logic, with the accompanying restrictions. For example, meta-level knowledge (i.e., knowledge about knowledge) is not rigorously supported by this representation. It is possible to define a state feature that indicates that some other state feature's value is known (e.g., the feature/value pair **(know-F T)** could mean we have a valid value for the feature **F**), but the meta-level linkage between the two is not supported directly by the representation or inferencing mechanisms; instead, the system designer is responsible for building RTS primitives that would maintain the appropriate relationship between **(know-F)** and **F**. CIRCA does not yet implement a general meta-level operator which would take a first-order predicate (feature) as an argument, and return the status of the system's knowledge of that feature.

Pure logic also lacks the ability to represent metric values and time. The AIS world model has addressed time representation in an unusual fashion, using only worst-case timing values for the duration of temporal transitions and for transition $min\Delta$ values. This rudimentary representation of time supports only one temporal relation between transitions: preemption. However, as only worst-case behaviors are of interest in building a real-time plan that is guaranteed to achieve its goals, preemption (and thus the ability to avoid failure) is the only crucial aspect of time that must be modeled. More complex relations, such as the overlapping-interval relations defined by Allen [3], are not available to the control-planning world model[3].

Because the AIS must enumerate all possible world states, continuous-valued variables are a problem: if all their values are possible, the state space is infinite. In general, however, this limitation has proven quite reasonable, because the TAPs that are being planned provide only a very discrete-valued type of service: either a TAP is fired, or it is not. Making this decision does not require the full power of continuous variables. Essentially, making this boolean decision is simply a matter of applying a threshold to a continuous value. That numeric operation can be abstracted by the system designer (who is encoding the world model) to yield a discrete variable (state feature) suited for use in the world model, even if the actual implementation of the TAP primitive on the RTS will use continuous variable computations to derive the value of the discrete feature.

For example, in the Puma domain, there is no need to include in the world model a state feature representing the distance to a part arriving on the conveyor belt, and then use that continuous-valued feature to decide whether the robot can grasp the part. Instead, the above thresholding technique can be applied to derive a related, boolean variable indicating whether the part is in range. The abstraction away from the continuous-valued, physical model makes the state space feasible.

This type of abstraction has a parallel effect on the representation of the world model transitions. Rather than including a transition that specifies the mathematical changes to continuous-valued state features, transitions can simply specify what possible changes to the discrete-valued abstract features are possible. For example, again in the Puma domain, the transition that indicates that a new part may arrive is not represented as a mathematical

[3]Although it should be noted that such powerful techniques have been used by the AIS in the higher-level planning mechanism, when non-worst-case timing considerations may be of interest.

function of conveyor speed and part separation. Instead, a simple temporal transition is used to show that, after the last part arrives, the next part may arrive after some minimum delay. That delay may be automatically computed by the higher-level AIS mechanisms, or it may be fixed by the system designer.

The AIS world model implements a very simple form of uncertainty representation via nondeterministic transitions: there is no explicit knowledge of probabilities or other bias in uncertainty. The motivation for this simplicity is, again, the need for guaranteed real-time control plans. To make true guarantees, the system must consider all possible world behaviors, no matter how unlikely. Therefore, measures of uncertainty have no use, at that simple level. However, as we move beyond the simplest model of CIRCA's operation, and investigate its ability to make performance tradeoffs, it is clear that a more explicit understanding of the relative likelihood of different events would be useful in deciding what types of tradeoffs should be made. In Chapter 7 we will discuss the performance tradeoffs the AIS can make, and postulate some motivations for these tradeoffs, based on uncertainty information that the world model currently does not include.

The AIS makes an additional assumption that the world model descriptions are complete: all features of the world must be represented explicitly, and likewise all possible changes to state features must be represented as transitions. These completeness requirements are needed to ensure that the AIS is able to reason about all possible sequences of events in the environment, so that all worst-case behaviors can be predicted and planned for. If completeness is not possible, the system can still build plans, but they will only be guaranteed to deal with the modeled portions of the environment's true state space. See Section 3.4 for more discussion of the meaning of guarantees based on partial models. Section 7.2 illustrates some of the results of operating with an incomplete world model.

4.14 Summary of Agent/Environment Characterization for Guarantees

In this chapter we have described CIRCA's world model; in the process, we have identified critical pieces of information that an agent needs to model in order to make guarantees about its performance in its environment. These characteristics of 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; sensors must have predictive sufficiency (as discussed in Section 4.12).
- 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.

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 the 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.

CHAPTER 5

THE AI SUBSYSTEM IMPLEMENTATION

In developing a prototype implementation of the CIRCA architecture, we have focused on three main contributions of the design. First, the prototype system includes a carefully crafted subsystem interface designed to bridge the gap between the uncertain performance of the AIS and the rigid constraints of the RTS. Second, the prototype AIS includes a planning system which develops TAP plans to provide control-level guarantees based on the graph model described in Chapter 4. Third, when resource restrictions make ideal performance impossible, the prototype AIS has several strategies for reducing resource requirements, trading off the various dimensions of performance against each other.

In this chapter we examine the implementation details of the prototype AI subsystem, paying particular attention to the control-level TAP planner, which constitutes the main body of the CIRCA code development. The performance tradeoff methods used by the AIS are described in Chapter 7, in the context of evaluating the prototype system's capabilities.

5.1 Overview of the AIS

The design of the prototype AIS is motivated by several operational requirements:

- **Representation:** The AIS must be able to represent and reason about descriptions of the system's environment, descriptions of its primitive capabilities, multiple task-level and control-level goals, long-term plans, and TAP plans.

- **Reasoning:** As described in Chapter 3, the AIS is responsible for reasoning about the overall CIRCA system's operations, so it must consider both its own activities and the behavior of the RTS and Scheduler. Thus the AIS must have meta-level reasoning capabilities allowing it to perform deliberation and planning at levels above the base domain level.

- **Communication:** To send TAP schedules to the RTS and receive feedback information, the AIS must have mechanisms for I/O. CIRCA is intended to operate in dynamic environments where goals may change and feedback information may require TAP plan modifications, so the AIS should be able to respond to feedback information by modifying its problem-solving behavior.

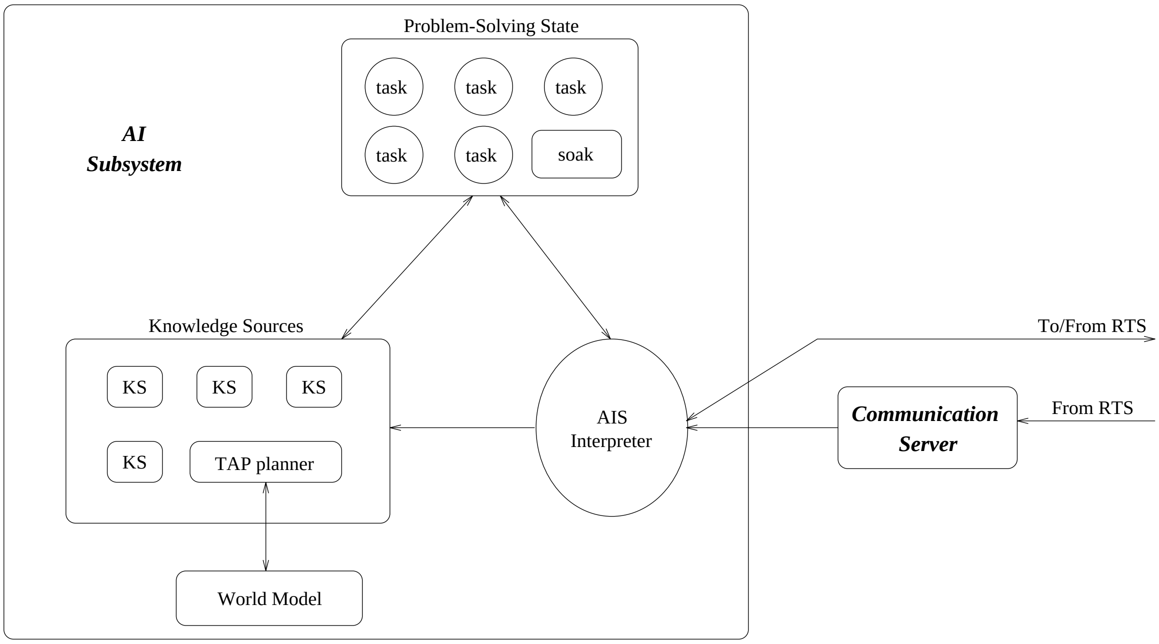


Figure 5.1: Conceptual schematic of the prototype AIS.

In the prototype implementation, illustrated in Figure 5.1, the deliberative processing used to build TAP plans is implemented separately from the more general, higher-level deliberative mechanisms used to build long-term plans, make performance tradeoffs, and control the communication processing of the AIS. The TAP planner is a code module that is simply invoked by the higher-level processing mechanism, which is cast as an interpreter. After describing the interpreter in the next section, we describe the TAP planner and its representations in Section 5.3.

5.2 The AIS Interpreter

The main purpose of the AIS interpreter is to build long-term plans to achieve the system's goals, and then break those plans into sequences of subtasks that can be implemented by TAP plans. The interpreter then invokes the TAP planner module with the appropriate goals and domain description, so that it generates a TAP plan that will achieve the goals of each subtask.

The problem-solving state of the AIS interpreter is stored in a dynamic set of data structures called *tasks*, where each task structure includes information describing a problem to be addressed. For example, when a feedback message from the RTS is waiting on the AIS' socket queue, the AIS represents the required task of reading in that message by an **RTS-msg-task** structure whose **status** slot is set to **'waiting**.

The AIS interpreter processes these tasks by running Lisp code structured into Knowledge Sources (KSs) that are similar in form to those of a blackboard system [58]. Each KS has a set of class-constrained variables, a parameterized precondition expression, and an action expression that is executed if the KS is actually "fired." For example, the simple

```
KS read-in-RTS-msg
   :VARIABLES ((RTS-msg-task-p ?task))
   :PRECONDITION (eq (status ?task) 'waiting)
   :ACTION (read-in-RTS-msg ?task)
```

Figure 5.2: The simple **read-in-RTS-msg** KS.

read-in-RTS-msg KS is shown in Figure 5.2. The `VARIABLES` declaration binds the `?task` variable to a task structure of the appropriate class. The `PRECONDITION` expression tests to make sure that the incoming message is in the correct state, and the `ACTION` expression invokes a procedure to read the message in from the socket.

To determine which KSs are applicable to the current problem-solving state, the KS variables and preconditions are matched against the task structures via a unique Rete net [17] implementation. The Rete implementation allows the preconditions of each KS to apply arbitrary Lisp tests to the KS variables, allowing very powerful expressions describing when different KSs are applicable. The KS precondition is formed from either a single expression or a list of expressions that are implicitly conjoined. The Rete system invokes the Lisp interpreter on each of the conjuncts using `(eval)`, and retains partial match information to speed the match process. The flexibility of this precondition representation incurs the cost of relatively slow execution: Rete systems using more restricted precondition languages can be highly optimized.

The interpreter mechanism that chooses the next KS to fire is drawn almost directly from the PRS architecture [23, 31], and bears little resemblance to a blackboard's agenda mechanism. Figure 5.3 shows a slightly simplified version of the Lisp code for the prototype AIS interpreter. Each cycle of the interpreter finds the set of KSs whose precondition expressions are true in the current problem-solving state (the "set of applicable KSs" or **soak**), and then asserts the value of the current **soak** into the Rete net, essentially making the problem-solving state represent the fact that the system is considering executing those KSs. This new assertion may cause new, meta-level KSs to match in the Rete net. The meta-level KSs are responsible for choosing which KS to fire from the **soak** matched in the previous interpreter cycle.

Figure 5.4 shows an example meta-level KS that chooses which lower-level KS to fire for a particular task, based on a global strategy variable (that some other KSs can manipulate). When any KSs match at the meta-level, the interpreter removes the previously asserted **soak** from the Rete net and asserts the value of the new (meta-level) **soak** to be exactly the set of newly matched, meta-level KSs. Again, new KSs may match against this assertion, forming a meta-meta-level. In this way, the interpreter can climb an arbitrary number of meta-levels. When no new meta-level KSs match, the system executes a single KS, chosen randomly from the **soak** of the previous reasoning level.

The prototype AIS interpreter differs from PRS in the relatively unstructured form of our KSs, and the lack of an architectural "intentions" structure. In the prototype AIS, firing a KS simply means running some block of Lisp code. A PRS Knowledge Area (KA), on the other hand, is a structured representation of the set of plans to achieve a goal.

```
(defun AIS-interpreter ()
  (assert-initial-world-model)
  (setf *soak* (get-all-matched-KSs))
  (while T
    (assert *soak*)
    (setf *new-soak* (get-new-matched-KSs))
    (cond ((and (null *soak*) (null *new-soak*)) ;; Stop if no KSs matched.
           (return))
          ((null *new-soak*)                     ;; If no meta-level KSs
           (execute-KS (random-choice *soak*))   ;; matched, fire one from
           (unassert *soak*)                     ;; last soak.
           (setf *soak* (get-all-matched-KSs)))
          (T (unassert *soak*)                   ;; Else, go to meta-level.
           (setf *soak* *new-soak*)))))

(defun bootstrap-AIS ()
  (setup-signal-handlers)
  (catch 'terminate
         (setf *soak* (get-all-matched-KSs))
         (while T
                (catch 'interrupted
                       (AIS-interpreter)))))
```

Figure 5.3: The prototype AIS interpreter.

```
KS strategy-choice
   :VARIABLES ((task-p ?task) (soak-p ?soak))
   :PRECONDITION (get-KSs-for-task ?soak ?task)
   :ACTION (let ((task-KSs (get-KSs-for-task ?soak ?task)))
             (execute-KS (random-choice
                           (get-KSs-for-strategy task-KSs *strategy*))))
```

Figure 5.4: The **strategy-choice** meta-level KS.

When a PRS KA is chosen by the interpreter described above, it is merged into the PRS intentions structure, which represents the cognitive commitments of the system. The KA is then executed at some later time, as the intentions structure is traversed by the PRS execution phase (which has no parallel in our implementation). Cognitive commitments are represented in our system by task objects, which are manipulated by KSs in the same way as other tasks, rather than by architectural mechanisms.

5.2.1 Interrupt Handling

Although the AIS is never constrained to meet deadlines, we would like it to respond quickly to changes in goals or RTS feedback, so that it allocates its deliberation resources to the most important current task. Because the AIS interpreter may invoke complex, time-consuming processing that would make its worst-case response time unacceptably long, it is built to be interruptible. The interpreter installs two customized signal handlers in its Lisp environment, to handle timeouts and communication interrupts.

Timeouts

Timeouts are used to limit the amount of AIS processing that is committed to any particular task whose problem-solving methods may have uncertain or high-variance processing requirements. The AIS can run this type of task with a limited time allocation by first forking off (in the true Unix sense) a simple timer process that will send a `SIGUSR1` signal to the AIS Lisp process after a fixed amount of time (set by the AIS). The AIS invokes the task processing after forking the timer. If the task processing takes too long, the forked timer will send the signal, and the AIS will be interrupted. The signal handler for the timeout signal creates a new **timed-out-task** structure, adds it to the Rete net, and then passes control back to the base-level AIS interpreter using a **(throw 'interrupted)** call (see the **(catch 'interrupted)** in Figure 5.3). This has the effect of terminating the processing of the current KS and giving the AIS interpreter the opportunity to re-examine the state of the system, deciding whether to execute the interrupted KS again or deal with the timeout in some other manner, possibly by making compromises that simplify the required task processing.

For example, the process of building a TAP plan is implemented by a KS that keeps track of its progress in a set of global state variables. If the TAP-planner KS is interrupted, it can resume processing on the next KS execution at (nearly) the same point at which it was stopped, by examining this global state (for more details, see Section 5.3). Alternatively, other KSs may intervene and alter some aspect of the TAP planner's state, so that the planning operation will be simplified or altered entirely. Several examples of these sorts of changes, and the resulting performance tradeoffs, will be discussed in Chapter 7.

Communication Interrupts

Feedback data from the RTS can arrive at unpredictable times and may indicate high-priority changes in the environment which require the AIS' attention. Thus we would like the AIS to be interrupted by feedback from the RTS. However, because the RTS runs on a different processor, it cannot send an interrupt or signal directly. Instead, the AIS must make special provisions to handle incoming messages from the RTS. When the AIS initializes, it opens a socket on a known port address for the RTS to connect to. The AIS also forks off a simple communications subprocess (COMM), written in C, which also opens a socket for the RTS. When the RTS is initialized, it connects to both the AIS and the COMM sockets. Whenever the RTS sends a message to the AIS over their socket link, the RTS also sends a short "wakeup" message to the COMM process. COMM simply loops repeatedly around a test that waits for that wakeup message, and then sends `SIGUSR2` to the AIS Lisp process[1]. In response, the appropriate AIS signal handler will create a **RTS-msg-task** with a **status** of **'waiting**, meaning that there is a message waiting to be read in from the RTS. As with timeouts, the signal handler then **throw**s back to the interpreter, which can choose whether to read in the message from the AIS socket or leave the message waiting while the AIS performs other, more important task processing.

This somewhat complex arrangement has the effect of allowing the AIS to remain highly alert to incoming RTS feedback without constantly polling its sockets. This means that the code used in KSs need not be strictly limited in runtime, or interspersed with polling calls. However, because the normal interrupt handlers do not return control directly to interrupted KSs, the KSs do need to be written carefully when they make changes to stored information. To maintain a consistent processing state, some KS operations must be atomic— that is, they must either be run to completion, or not be started.

Critical sections of KS code that must be atomic are built as "monitors," protected from interrupts by calls to **(begin-monitor)** and **(end-monitor)**. The **(begin-monitor)** call simply replaces the signal handlers with modified versions that return to the interrupted KS after setting a global flag indicating that an interrupt has arrived. The **(end-monitor)** call restores the original signal handlers and also calls the appropriate signals handlers if the global interrupt flags have been set during the preceding monitor code.

Remaining interruptible gives PRS and our AIS the useful ability to perform arbitrarily complex computations within a KS while also attending to ongoing world events. In particular, Ingrand and Georgeff [31] have shown that, given certain reasonable assumptions about event frequency and KS precondition complexity, the prototype AIS will notice every event that generates an interrupt. While our AIS has the ability to implement arbitrarily complex processing during both the matching and execution of KSs, the current experimental domains have only used limited procedures that do meet the requirements for bounded reaction time. However, the Lisp and Unix basis for the AIS makes rigid response bounds impossible.

5.2.2 Existing Knowledge Sources

Because the main focus of this work has been on building guaranteed TAP control plans using the world model described in Chapter 4, the knowledge built into the AIS interpreter has not been extensively developed. The current KSs for the Puma domain are used largely to manage and interleave the processing of feedback messages from the RTS with invocations

[1]The COMM process can send a signal to the AIS because it is running on the same processor, unlike the RTS.

of the TAP-planning KS and the process of downloading new TAP plans to the RTS. KSs to automate the performance tradeoffs described in Chapter 7 are under development.

In the previous prototype implementation of CIRCA, which was applied to a mobile robotics domain, the AIS interpreter played a much larger role. In that domain, a Heathkit Hero 2000 robot navigated through hallways under the control of TAP plans, and the AIS planned paths and built TAP plans to implement the appropriate navigation strategies. That version of the AIS had KSs to incrementally form hierarchical navigation plans, given a building map, a destination, and the current location of the Hero. It also had KSs that implemented a primitive form of interval temporal reasoning capable of propagating ordering information. These KSs allowed the system to prioritize the hierarchical decomposition of path plans, so that TAP plans were generated first for the earlier portions of the planned robot path.

The AIS also had KSs that implemented two forms of performance tradeoffs when the domain was overconstrained. If the AIS recognized that collisions with obstacles were possible because of limitations on the robot's sensing speed, a KS could slow down the robot's forward motion until safety could be assured. Alternatively, a KS could sacrifice the guarantee on the TAP which used a sonar sensor to check the distance to the walls, making sure that the robot was moving down the middle of the hallway. By removing that TAP from the list of required TAPs, the sensing needs of the system were reduced, and obstacle collisions could be avoided. However, because the wall-checking TAP was only being executed in a best-effort fashion, the robot was no longer guaranteed to avoid colliding with walls.

Overall, the prototype AIS interpreter has proven to be a very flexible platform for controlling deliberation, particularly because of the ease with which the system climbs to meta levels to decide among competing processing demands. The interrupt mechanisms allow the system to remain alert while still implementing complex planning functions. The use of arbitrary Lisp code in both the preconditions and actions of KSs makes it possible to implement complex KS behaviors without the contortions imposed by less-flexible representations. On the other hand, this also means that automatically parsing KS preconditions is difficult, and the AIS Rete implementation is not particularly fast.

5.3 The TAP Planner

The main use of the AIS interpreter in the Puma domain is to coordinate invocations of the TAP planner module, which instantiates and reasons about the world model (described in Chapter 4) to develop TAP control plans. The TAP planner essentially performs the design phase shown in the Figure 3.2 flowchart of automated real-time system design in Section 3.5. 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 mechanisms possible.

From the description of the world model in Chapter 4, 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 is given a description of the goals that a particular TAP plan should achieve, a description of the possible transitions in the world model, and a set of initial states. The planner builds up a list of actions that will achieve its goals and will also restrict the agent to a safely-controlled set of states surrounding the initial states, by making failure states in the world model unreachable and by preventing the world from leaving the safely-controlled set of states. This planning 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 a state machine with global, explicit state storage, as illustrated in Figure 5.5. 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 computes its decision, pushes the alternative choices for that decision onto a *choice-stack*, sets the mode variable

```
(defun run-TAP-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 5.5: The main loop for the AIS TAP planner.

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 `T`. 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). If the system tries to backtrack off the end of the choice-stack, this is an indication that the planner has failed to find a plan, and some modifications to the design specifications will be necessary, as diagrammed in Figure 3.2.

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 state of this implementation also has an advantage over recursive implementations, because in this formulation it is fairly easy to interrupt and resume the TAP planner.

5.3.1 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 [59]. Figure 5.6 shows example rules from the Puma 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.

Initial state descriptions currently must specify all features of the world, as illustrated

```
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) (part-in-gripper nil))
      POSTCONDS: ((emergency nil) (robot-position over-button))
      RESOURCES: (arm gripper)
      WCET: 3.5 [seconds]
```

Figure 5.6: Example transition descriptions given to the AIS.

in Figure 5.7. Recall that each TAP plan is being built to achieve some goals that are part of a longer-term sequence of steps determined by the AIS interpreter. Thus the AIS chooses the initial state description and goal description for each TAP plan according to its position in the long-term plan. In the future, it might be useful to allow the AIS to specify only partial initial state descriptions, indicating that the new TAP plan might be started in any of several states achieved by the previous TAP plan. It would be straightforward to have the TAP planner enumerate the set of possible initial states I before beginning the planning process.

Goal descriptions do not usually specify the entire state of the desired world: in fact, many describe just a single feature (such as **(part-in-box T)**). These partial descriptions are not expanded into an explicit set of acceptable states; instead, the AIS uses the descriptions as litmus tests for states which it generates on-the-fly, as detailed below.

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 [59]. 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. The flowchart in Figure 5.8 illustrates the planning algorithm.

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[2], the AIS will ignore it and pop the next state off the stack. If the current state is reachable, the AIS

[2] 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.

```
INITIAL-STATE
    FEATURES: ((failure nil)
               (emergency nil)
               (know_type_of_conveyor_part nil)
               (know_type_of_table_part nil)
               (part_in_gripper nil)
               (conveyor_status free)
               (robot_status free)
               (robot_position over_table)
               (part_on_table nil)
               (part_on_conveyor nil)
               (part_in_box nil))


GOALS:   ((part_in_box T)
          (part_on_conveyor nil)
          (part_on_table nil)
          (part_in_gripper nil))
```

Figure 5.7: Example initial state and goal descriptions given to 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 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 actions the one that leads to the best next state, as determined by a heuristic scoring function (described below in Section 5.3.1). 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 4.11). 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

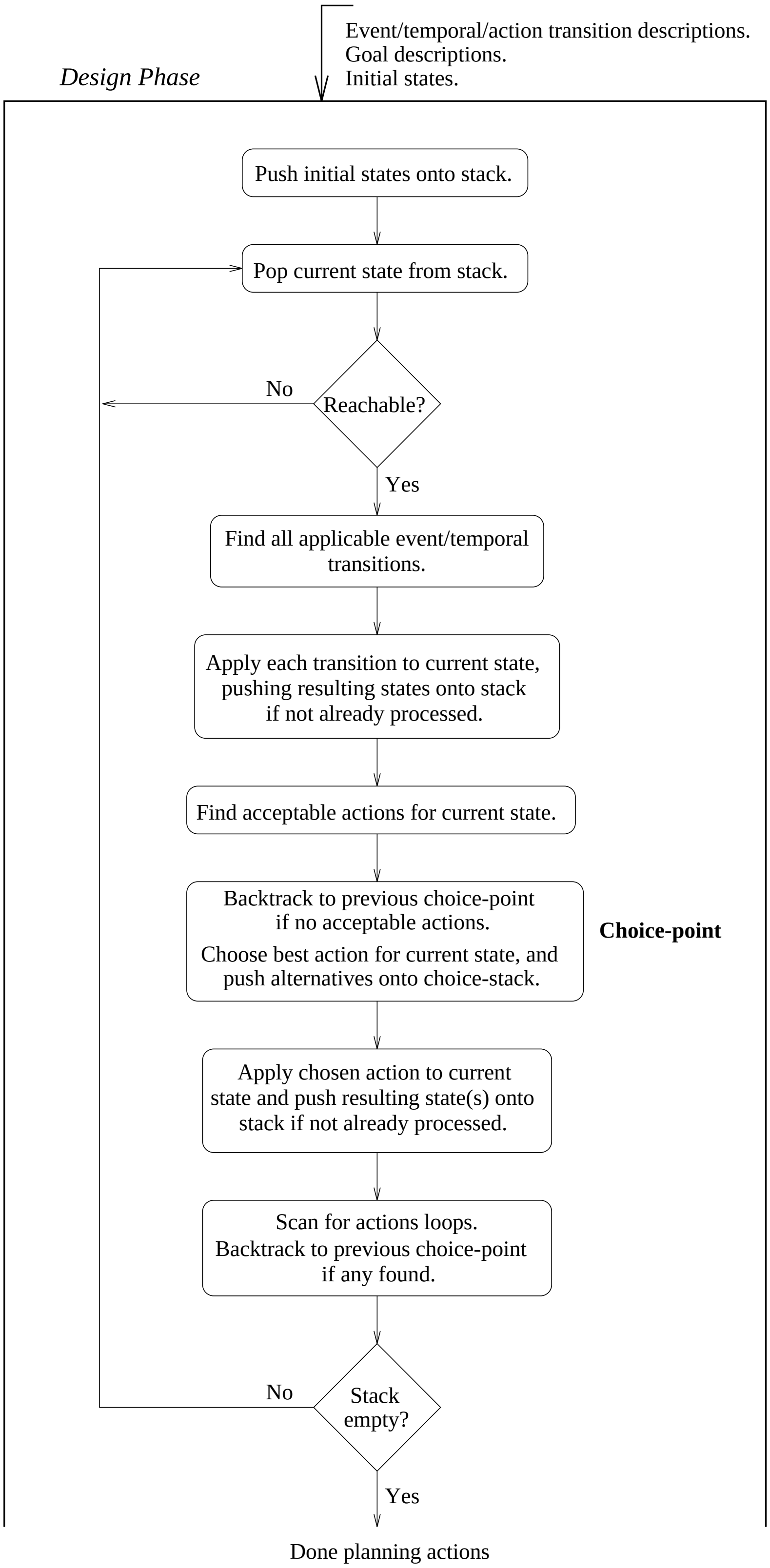


Figure 5.8: A flowchart for the action-planning algorithm.

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 acceptable action at each state, the AIS can perform a complete search of the set of action plans.

Complexity

We noted in Chapter 1 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 [76]. 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 the 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.

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 [24] 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.

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 [57].

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 [41, 54] 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 5.9, is placed in the loop shown in Figure 5.5, 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 does not 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 5.5 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 any-time planning algorithm [9, 65]. 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 the Puma domain, the system implements the control-level goal of making sure that nothing

```
(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 5.9: A decision function implementing an incremental improvement method.

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 parts sitting on the table, the faster the system will achieve its task-level goal of generating a packed box.

The Scoring Heuristic

The scoring function used to choose actions is the only heuristic knowledge currently used by the TAP planner. The scoring function performs a recursive N-step lookahead, finding and returning a value corresponding to the best state reachable in N transitions from the current state. Based on this analysis, the control-level planner chooses to perform, for each state, the action which leads to the best scoring state. Note that backtracking may lead the system to make alternative choices if the initial choice leads to a failure.

To find the score for a transition applied to a given state with x-step lookahead, the scoring function simulates the proposed transition to build a description of the resulting state. The function then derives and saves a score for the value of that state. If $x > 0$, the function then finds all the transitions which may apply to that state, and recursively calls itself to find the score for each of those possible transitions when applied to the new state, with $x - 1$ steps of lookahead. The results from those recursive calls are saved, and the function returns the best score of among all those saved. If $x = 0$, no recursive calls are made, and the function simply returns the score of the state resulting from the application of the transition.

The current heuristic scoring mechanism considers several factors. First and foremost, 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. At each level of recursive lookahead, which corresponds to following an additional transition, the scoring function adds a small penalty to its resulting score, so that the value of a particular state is degraded partially by the "distance" separating it from the current state, as measured by the number of transitions between them. In the future, a more useful measure of the cost of a transition path might take into account the associated delays as well as the cost of the agent's actions.

One final consideration is necessary to choose correct actions in the Puma example. The representation of the Puma domain is encoded at a fairly high level of abstraction, to avoid excessive detail and the accompanying state-space explosion. We will discuss this topic more fully in Section 5.4; for now, it is sufficient to note that we do not want the domain model to include a feature that would be subject to "counting." That is, a domain feature expressing *how many* parts are in the box, or *how many* parts are on the table, would be a very unfortunate choice, because it would lead to a potentially infinite (or at least very large) search space, as the planner would have to reason about a complete set of world states where one part was in the box, and another set where two parts were in the box, etc.

So instead, we encode the world with a boolean feature that simply indicates whether a part is in the box or not. And the goal, in such a domain, is to achieve states where **(part-in-box T)** holds. But now consider what happens when the planner reasons about states where the robot has already placed a part in the box, and it is now processing another part. The goal **(part-in-box T)** already holds, so there is no scoring differential to encourage the system to place the new part in the box as well.

To deal with this sort of difficulty, CIRCA also allows the system designer to designate *repeat goals*. These are goals which it is valuable to achieve again, even if they are already true. With this consideration added to the scoring function already described, the TAP planner is able to choose intuitively correct actions in all situations that arise in the Puma domain (given sufficient lookahead). In the current domain implementations, a lookahead value N of four is sufficient to allow the planner to always choose the correct action.

5.3.2 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 [34]. 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.

For example, in the Puma domain, the planner initially plans to take the action **push-emergency-button** in 54 states, each of which has eleven features. After minimization, the action is associated only with tests for **((emergency T) (part-in-gripper nil))**. The new tests do not check all eleven state features 54 times each, so they will take much less time to execute. Of course, with only those two preconditions, the resulting TAP will match many more than the originally planned 54 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 5.6 included the precondition **(robot-status free)**, but that precondition was removed during minimization because it is not needed to distinguish the 54 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[3] [62], 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.

5.3.3 Planning Sensing

Once the action preconditions have been minimized, the AIS plans sensing actions to implement the TAP test expressions. 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

[3] Marcel Schoppers suggested this approach.

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

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

Figure 5.10: Example sensor and virtual sensor descriptions.

execution time. Figure 5.10 shows two example sensor descriptions.

The first example describes a physical sensor in the system, the overhead camera that returns information about part shapes 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?** 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 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 can 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 can be built that will yield a sufficiently short TAP, then the backtracking propagates back to the previous processing phases, and the system would search for a different control plan.

The sensor-planning functionality, as described above, has not been fully implemented in the Puma domain, but could be easily added to the modular TAP planning loop as another decision function. The search-based sensor-planning function would operate in much the same manner as the action-planning function, choosing a sensor mapping on each invocation, and backtracking by simply returning `NIL`.

The sensor-planning phase has been implemented in the Puma domain in a slightly simplified form. The simplification retains the essential purpose of the sensor-planning concept, which is to allow CIRCA to map abstract world model state features to different real-world features. For example, the world model feature **know-type-of-conveyor-part** indicates whether CIRCA knows how to pack the particular shape of the current part on the conveyor into the box. Because the set of "known" part shapes changes over time, as CIRCA derives packing methods for more and more parts, the precise meaning of the **know-type-of-conveyor-part** feature changes over time. This change must be propagated into the TAPs executed by the RTS, so that the TAPs act appropriately and use the new packing methods. Thus the mapping of this world model feature to actual tests of real-world features must change.

During the sensor-planning phase, the AIS currently uses an association list to map world model features to combinations of detectable, real-world features. The association list essentially bypasses the search processing that might otherwise be used to perform sensor planning. The association list can be changed by the same KSs that derive new knowledge for the system, such as the KSs that figure out how to pack new part shapes.

For example, when the system is first introduced to the Puma domain it is only told how to pack square parts into the box, so the sensor-planning phase maps a TAP precondition of **(know-type-of-conveyor-part T)** into the test **(type-of-conveyor-part 'square)**. Likewise, a TAP precondition of **(know-type-of-conveyor-part nil)** is mapped to **(not (type-of-conveyor-part 'square))**. After the system has seen a rectangle arrive and has developed a new packing method for both squares and rectangles, the association list is changed so that the abstract feature is mapped to a different expression: **(or (type-of-conveyor-part 'square) (type-of-conveyor-part 'rectangle))**.

Thus this feature-mapping mechanism is useful for translating abstract domain descriptions into more realistic, dynamic conditions on the environment. This type of variable sensor planning is crucial both to implementing abstract descriptions of the environment, and to the flexible use of multiple sensing modalities.

5.3.4 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.

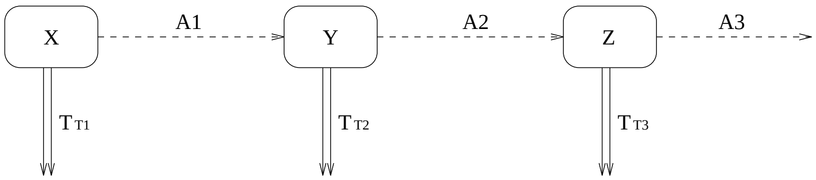


Figure 5.11: Example actions dealing with dependent temporal transitions.

For TAPs dealing with dependent temporal transitions, the problem is complicated by the dependencies between TAP periods. For example, Figure 5.11 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 can 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[4]. 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:

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

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

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

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 5.11, 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. Improved TAP period assignment algorithms could prove a vital area of future work, of particular interest to scheduling and real-time systems researchers, who usually assume that these periods are pre-determined.

5.3.5 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. Chapter 6 provides complete details on the scheduling algorithm currently implemented. If the Scheduler cannot build a successful schedule to guarantee all the TAP timing constraints, it will return a failure message to the AIS. At that time, the AIS may backtrack to generate a different proposed TAP plan, or it may make other alterations to its world model to trade off some aspect of its performance, in an attempt to relax the scheduling constraints that made a TAP schedule impossible to find.

5.4 Discussion

Our goal in developing CIRCA's AI Subsystem was not to build the "ultimate planner," but rather to investigate the requirements for building guaranteed real-time control plans. As a result, the AIS implementation we have developed is not highly optimized. Instead, we have focused our attention on the reasoning and representation capabilities that allow the prototype AIS to build plans with well-understood temporal behavior. The following discussion of the AIS' TAP planner is thus focused on describing the assumptions, strengths, and weaknesses of the system's representations and reasoning mechanisms.

To begin this discussion, it will be helpful to examine CIRCA's approach to the representation and reasoning issues in a common domain called the Wesson Oil Problem.

5.4.1 Cleaning Up the Wesson Oil Problem

Research into reactive systems has just begun to develop a set of standardized problems which can be used to compare systems. From the perspective of intelligent real-time control, one of the more interesting new benchmarks is the Wesson Oil Problem (WOP), as described by Gat [21, p. 40]:

> The name derives from a television commercial for Wesson Oil in which a housewife is frying chicken (in Wesson Oil, of course) when one of her children suddenly falls down and has to be taken to the hospital. However, before going to the hospital the housewife turns off the stove...
>
> The action of turning off the stove... is an example of a *clean-up procedure* which is executed when a high-priority task (taking the kid to the hospital) interrupts a low-priority task (frying chicken).

Gat discussed the Wesson Oil Problem to motivate the need for clean-up procedures in his reactive ALFA language. Using a mechanism similar to the Lisp `unwind-protect`, he was able to protect low-priority tasks so that, when interrupted, a specially-tagged procedure was run to clean up their activities. This mechanism was used solely by hand-coded reactive plans.

Gat also noted that, ideally, cleanup procedures should be conditional on aspects of the world other than simply the fact that a low-priority process is interrupted. His example: "If a stranger walks into your kitchen with a gun, turning off the stove before running for your life may or may not be the right thing to do."

CIRCA's approach to this problem is unique in two ways. First, CIRCA does not simply permit users to design cleanup procedures, CIRCA actually automatically plans them itself. Second, CIRCA's planned cleanup procedures are fully context-sensitive; CIRCA will build different reactions to deal with the injured child and the invading stranger. To demonstrate these advantages, we now describe in detail CIRCA's reasoning about the WOP, as illustrated in Figure 5.12.

State $\mathcal{A}$ represents the initial state, where the housewife is peacefully cooking and the child is uninjured. The **oil-ignites** temporal transition to failure (state $\mathcal{F}$) indicates that, if the stove is left on too long, the oil will overheat and catch fire. For now we will ignore this problem, and consider what happens if the world follows the **child-falls** event transition to state $\mathcal{B}$. Reasoning about that state, the TAP planner will find there are two applicable actions: either the housewife can leave the kitchen to help the child, or she can turn off the stove. Looking ahead to the subsequent states, the scoring function will find that state $\mathcal{C}$, following the **leave-kitchen** action, still has a temporal transition to failure, while state $\mathcal{D}$ following the **turn-off-stove** action, does not. Therefore, the scoring function will prefer the latter option, and CIRCA will plan a TAP that implements the **turn-off-stove** action immediately after the child is hurt. Thus CIRCA has automatically planned a cleanup action to preserve the system's safety.

To demonstrate that the cleanup action planning is context-sensitive, consider the **stranger-arrives** transition from state $\mathcal{A}$ to state $\mathcal{G}$. In this case, the **(emergency T)** world feature is used to represent the armed stranger's threat, and the short $min\Delta$ of the **stranger-shoots-housewife** temporal transition to failure indicates that the situation is highly urgent. Reasoning about this situation, the TAP planner will again project the two possible actions **leave-kitchen** and **turn-off-stove**. Presumably, the threat from the stranger is so dire that the housewife will not have time to turn off the stove (or else this situation is no different from the previous case). Thus, the planner will recognize that it cannot choose the **turn-off-stove** action, because then failure would be possible due to the

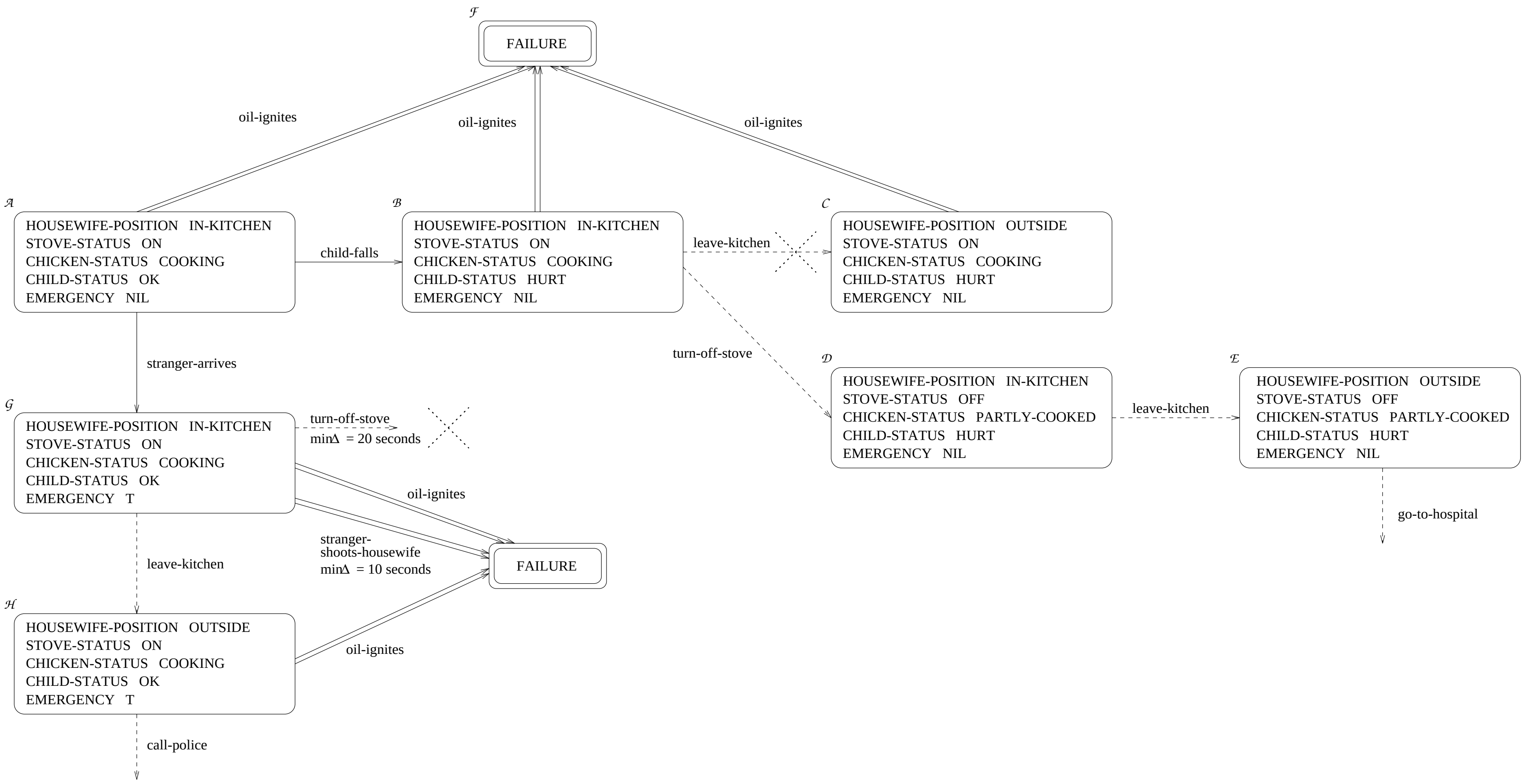


Figure 5.12: The Wesson Oil Problem world model.

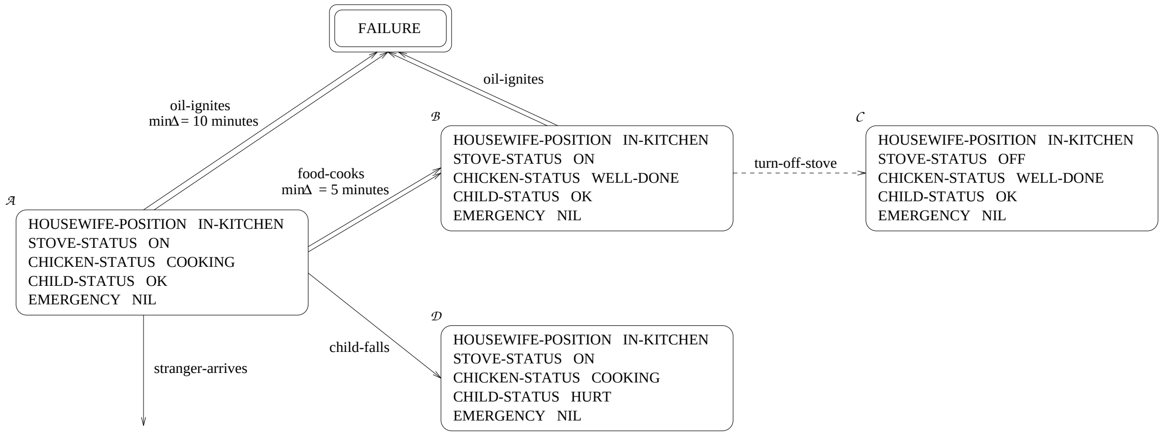


Figure 5.13: A portion of the modified Wesson Oil Problem world model.

stranger firing. Therefore, the planner will eliminate that possibility and choose to leave the kitchen instead.

Thus CIRCA's planning behavior results in two different TAPs, one that detects appropriate situations to turn off the stove, and one for leaving the kitchen. CIRCA has automatically derived reactions that clean up ongoing processes, but only when appropriate. While this example has demonstrated the strength of the CIRCA approach to planning situated cleanup actions, it also reveals a limitation of the world model and the representation of transitions.

5.4.2 The Transition Representation

We noted above that the **oil-ignites** temporal transition from state $\mathcal{A}$ means that failure may occur if the stove is left on too long. With just that transition represented, CIRCA would plan to turn off the stove immediately, thus preventing the fire. However, the stove must be left on for some period of time, in order to cook the chicken. An intuitive way to represent the required cooking time would be to use a temporal transition, as shown in the modified domain fragment in Figure 5.13. Unfortunately, this representation does not have the desired effect in the current CIRCA implementation. We would expect the **food-cooks** transition to preempt the **oil-ignites** transition, because of the respective $min\Delta$ labels of 5 minutes and 10 minutes. However, the current semantics of temporal transitions do not allow this: a temporal transition is not guaranteed to occur, so it cannot preempt any other transition. Thus the scenario shown in Figure 5.13, if actually given to CIRCA, would still result in a plan to turn off the stove immediately from state $\mathcal{A}$, because otherwise the failure due to fire would always be possible. CIRCA cannot yet represent non-atomic time-consuming activities that are guaranteed to occur; all guaranteed activities must be primitives, as discussed earlier in Section 4.3.

Another limitation of the current representation for world model transitions is the inability to include situation-dependent effects— the postconditions of a transition cannot contain variables referring the results of the precondition tests. For example, suppose that picking up a part from the table in the Puma domain leads to different loads on the robot

```
ACTION pickup-part
      PRECONDS: ((part-in-gripper nil) (part-in-reach T) (part-shape ?ps))
      POSTCONDS: ((part-in-gripper T) (arm-load (weight-from-shape ?ps)))
      RESOURCES: (arm)
      WCET: 2 [seconds]
```

Figure 5.14: An illegal transition, containing parameterized postconditions.

arm, depending on the part's shape. Figure 5.14 shows how we might like to represent such a transition, where the postcondition value of **arm-load** is determined by the binding of the part-shape variable **?ps** in the preconditions. CIRCA cannot yet deal with such transition forms, in part because they simply hide state space complexity— the parameterization allows the single representation of Figure 5.14 to act as many different transitions, depending on the binding of the **?ps** variable. CIRCA currently forces the system designer to enumerate those transitions manually; separate transitions would need to be encoded for each possible value of **?ps**. Extensions to the AIS planning algorithm to handle parameterized transitions are relatively straightforward, but the RTS and TAP execution mechanisms would require considerable modification to allow the dynamic specification of variables that are bound during the tests of TAPs and used during the actions.

5.4.3 The TAP Representation

The TAP representation was developed primarily as a simple model for reactive behaviors which could be automatically generated by CIRCA's reaction planning system. As such, it is not a fully-developed robot programming language like RPL [50] or ALFA [20], designed for humans building complex programs. Instead, the TAP mechanisms implement a "programmable production system," where the primitives used by the TAP "productions" are defined by arbitrary, user-produced C code (or, in an earlier version of the RTS, in Lisp). The RTS TAP execution environment provides only the `IF--THEN` conditional construct, as well as the boolean functions `AND`, `OR`, and `NOT`.

On the positive side, this simplicity means that it is easy for the AIS planner (or another system implementation) to build and manipulate TAPs that are directly executable by the RTS. Their structure is very simple, and there are no complex language elements (such as scoped identifiers) which would make parsing or altering the TAPs difficult. For example, the test-minimization phase discussed in Section 5.3.2 was implemented using generic ID3 code with the addition of a very simple filter that modifies the ID3 test-tree output to the TAP test format.

Another benefit of TAP simplicity is predictable behavior: because they have essentially only two activity modes (either the action is executed or it is not), they provide a basis for plan guarantees without complex reasoning about arbitrary programs. All the planning system needs to understand is whether or not the TAP will be executed. Likewise, this aspect leads to simple TAP timing characteristics that make it easier to account for a reaction's resource usage, and easier to implement if-time TAPs. If arbitrary programming constructs were allowed in the representation of reactions, the RTS might not be able to detect when a reaction is not going to use all of its assigned resources, and an if-time reaction may proceed. With TAPs, the boundary between test and action portions gives the RTS a simple indication of when to check the current resource usage and possibly fire if-time TAPs.

On the negative side, the same simplicity limits the power of the representation. Explicit loops, variables, `ELSE` clauses, and other common programming constructs are not available. However, it is important to note that the functionality of many of these constructs can be implemented at both higher and lower levels of the architecture. Loops, for example, can be implemented at a higher level by multiple TAPs or at a lower level by C-coded primitives. In fact, CIRCA frequently automatically builds loops of TAPs that will maintain some desired set of states. `ELSE` clauses can also be implemented by multiple TAPs checking complementary preconditions, or by C primitives with their own conditional branching.

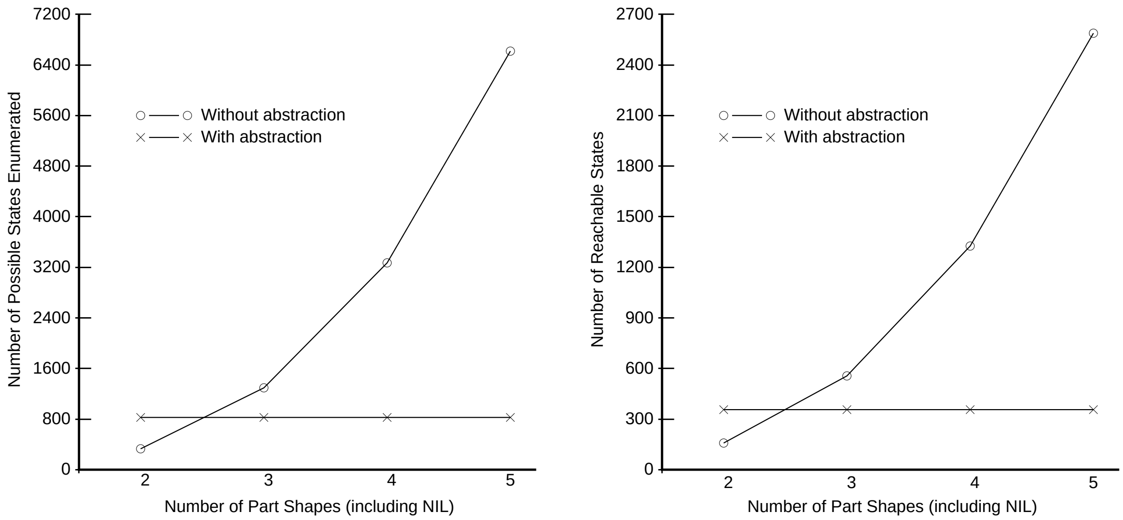
At issue, then, is not really absolute representational power of the TAPs executed by the RTS, but the balance between representational power at the planned-reaction, interpreted-code level and at the compiled level. At the C level, almost any programming constructs are possible. Those C constructs are compiled into primitives, which are then dynamically invoked by the TAP interpreter.

5.4.4 AIS Complexity and Abstraction for Domain Encoding

Although we argued in Section 5.3.1 that the world model planner will not enumerate the entire state space, it is unfortunately true that even the smaller set of "possible" world model states is still exponentially complex. For example, the Puma domain was originally encoded with separate state features indicating the shape of the parts held in the gripper, arriving on the conveyor, and last placed on the table. A `NIL` value for one of these features meant that no part was present in that location. When these three state features had only one possible non-`NIL` value (i.e., only parts of one shape were possible), the corresponding world model contained 330 possible states. Adding another possible part shape did not simply double the state space size, because of the need to enumerate each of the possible combinations of part types on the table, in the gripper, and on the conveyor. The resulting exponential growth of the enumerated state space is illustrated by the two graphs in Figure 5.15.

If we consider just the three part-shape features and their initial two possible values, we see there are $2^3 = 8$ possible combinations of values. Given the enumerated state space size of 330, we can see there are about $330/8 = 41.25$ states enumerated for each possible combination. If we use that scaling factor to project the number of enumerated states for larger numbers of possible part shapes, the results match quite closely with the experimental results. For example, if there are three possible values for the part shape features, we project that there will be $41.25 * 3^3 = 1114$ states; the actual state space contained 1294 possible states.

This exponential growth of the state space is highly undesirable because the time to search for control plans, while isolated from the real-time deadlines of the environment, nevertheless affects the speed with which the system can achieve its long-term goals. For example, while the AIS builds a TAP plan that can pack a new type of part, the RTS may



(a) Possible states.

(b) Reachable states.

Figure 5.15: The exponential growth of the world model state space.

have to halt the conveyor belt if it has filled up the table (buffer) capacity. In that case, the slow planning system is delaying achievement of the system's long-term goal of filling boxes, although the RTS is still guaranteeing the control-level goals of avoiding failure. In other words, while CIRCA does isolate the planner from the environment's control-level deadlines, there are still longer-term, non-critical timeliness concerns that motivate the desire for a system which builds plans more rapidly.

One way to address this type of exponential explosion is to use abstraction— we can reduce the enumeration of part-shape features by using more abstract values for those features. In the Puma domain, this approach was easily implemented by defining two abstract classes of part shapes: `KNOWN` and `UNKNOWN`. The system is assumed to have already derived a part-packing strategy for all `KNOWN` parts, while `UNKNOWN` parts must be put on the table until a suitable algorithm is derived. With these changes, the part-shape features have only three possible values, and the state space is reduced to 826 enumerated states, no matter how many different actual part shapes are possible. The state space in this case is smaller than the previous three-value case (1294) because the system does not know how to put `UNKNOWN` parts in the box. In the previous case, the box could be packed with two of the three "types" of parts (all but `NIL`), but the abstraction allows only one of the three types (`KNOWN`) to be packed.

Abstraction has the additional benefit of reducing the complexity of the transitions for the domain. In the Puma example, the original encoding method required separate transitions that applied to each of the individual part shapes, so that there was a **pickup-square-from-table** action transition, a **pickup-triangle-from-table** transition, etc. With the use of abstraction, these can all be compacted into a single **pickup-part-from-table** action transition, leaving the planner fewer transitions to match against.

5.4.5 Indexical Features and Looping

Another technique similar to abstraction proves useful in domain encoding to avoid enumeration problems that might result from individuating specific objects in the environment. As we shall see in a moment, the use of *indexical* features (or variables) [1] and nondeterminism also has advantages in the representation of repetitive agent behaviors.

Consider the problems that would arise in the Puma domain if the AIS planner attempted to reason about and distinguish between individual parts in its environment: i.e., if it assigned names to arriving parts and had to reason individually about **square21**, **rectangle13**, etc. Clearly the system would have to know all the possible parts that might arrive ahead of time, or else it would need the ability to generate new names as it postulates the arrival of new parts, and the state space would be infinite (what would stop it from continuing to postulate new part arrivals?). Even if the set of arriving parts is finite, the state space would still be vast, since each state would have to specify the position of each named part.

We have already seen the solution to this problem: rather than individuating parts, we must encode the environment using indexical features, which refer to objects by their relationship to our agent. For example, in the Puma domain there is a feature representing whether or not a part is held in the robot's gripper, but the specific name or identity of that part is never established. The only important information, from the agent's perspective, is the part's relationship to the robot. Thus indexical features abstract away from the identity of objects, but they do so in a slightly unusual fashion. For example, if **square21** is held in the robot's gripper, it will be affected by the actions referring to **part-in-gripper** (and CIRCA will never give it a name like **square21**). Later, that same part might be affected by actions referring to **part-on-table**. Thus the mapping of individual objects to their "classification" by indexical features is dynamic, changing as objects move through the world.

Indexical methods are frequently described in the control of reactive systems, but their use in planners is less common. CIRCA's combination of indexical variables and nondeterministic transitions leads to a uniquely powerful approach to planning repetitive and looping behaviors. Normally, planning looping behaviors (such as the Puma task of packing parts, hammering a nail, or driving a screw [52]) causes problems for planners because they reason about individual objects, and cannot recognize that they are building loops. For example, in the Puma domain, if a non-indexical planner first plans to deal with **square21** by picking it up, moving it over the box, and packing it in the box, each of those operators will have variables bound to the specific object (**square21**) being affected. If the planner later plans to perform the same actions on **rectangle14**, the actual representation of the plan operators will be different, because the variable bindings will be different. So recognizing a loop would require mapping back to general operators and comparing at that level. Even if this is done, it is still difficult to see how the planner would know when to look for a loop: when should it invoke the mapping and comparison functions, and on what portions of the current plan?

```
ACTION hammer-blow
      PRECONDS: ((arm-raised T) (in-gripper hammer))
      POSTCONDS: ( ((arm-raised nil) (nail-flush T))      ;; Either done
                   ((arm-raised nil) (nail-flush nil)) )  ;; or not yet.
      RESOURCES: (arm)
      WCET: .5 [seconds]
```

Figure 5.16: A simple nondeterministic transition that can be used to build dynamically-terminated plan loops.

Using indexical variables solves at least part of the problem in recognizing loops, because operators are not specialized with variable bindings. In our Puma example, we plan actions that deal with **part-on-conveyor** and **part-in-gripper**— separate operators are not planned or created for individual parts, and the fact that all arriving parts are picked up by the same repeated action is not derived by some inspection of the plan, it is already represented explicitly in the single planned operator.

We will see in a moment how CIRCA addresses the other part of the loop-planning problem: actually representing the loop. But first, we note that CIRCA's use of nondeterministic transitions also provides leverage on the problem. When a traditional planner is building a looping behavior, it may have particular difficulty deriving the termination conditions for the looping operator. If the loop has some known number of required repetitions, a simple counter can be used (as in NOAH). But what if the loop termination condition is dynamic, and cannot be precisely determined before run-time? Suppose, for example, that the task is to hammer a nail into a board, and uncertainty in the wood's density and the nail's shape does not allow us to predict exactly how many hammer blows will be required. How can a traditional planner with fixed, deterministic operators represent this? The effects of each hammer blow are not really certain; either the blow may finish the task by pushing the nail flush, or it may leave some part of the task undone, requiring additional repetitions. CIRCA has no trouble representing this uncertainty, as shown by the transition in Figure 5.16.

Planning a repetitive behavior in CIRCA is equally simple, as illustrated by the world model for the simple nailing domain, shown in Figure 5.17. In state $\mathcal{B}$, when the **hammer-blow** action transition is applicable, the planner will project forward both of its possible sets of postconditions, and will recognize that it may lead to the desired state $\mathcal{C}$, where **(nail-flush T)** holds. Thus the operator will be chosen correctly to accomplish the task. Projecting forward the other branch of the nondeterministic postconditions, the planner will also realize that the action transition may loop back onto state $\mathcal{A}$, which the planner has already considered. Since the planner has already selected an action for that state, no further planning is necessary. Thus CIRCA can easily plan looping behaviors with dynamic termination conditions that are determined only at run-time.

The looping itself, the repetition, is inherent in all the control plans that CIRCA builds, because they are implemented not as traditional sequential plans but as reactive TAP

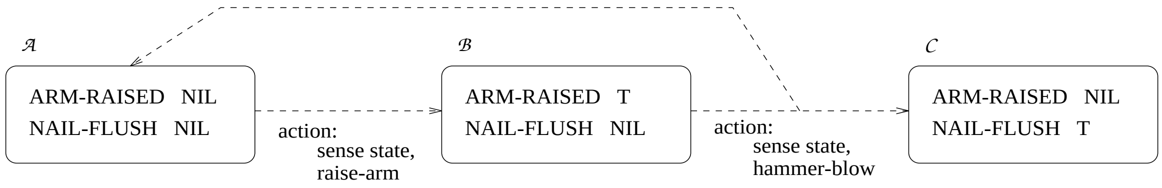


Figure 5.17: The nailing domain world model, demonstrating nondeterministic transitions and looping.

plans. The RTS continually loops over the schedule of TAPs, repeatedly testing their applicability conditions and executing their actions whenever appropriate. Thus, if a world model contains a loop (i.e., the planner thinks the world may re-enter a state it has been in before), the TAP form of the control plan already ensures that the state will be recognized and appropriate action taken, if necessary. The planner does not need to perform any additional reasoning to accommodate repeated behaviors.

5.5 Summary of AIS Features

In sum, our AIS implementation satisfies the functional requirements useful for intelligently designing and controlling a real-time system, as presented in Section 5.1. The AIS includes the following features:

- Flexible, Lisp-based Knowledge Sources.
  - Unconstrained precondition expressions.
  - Unconstrained action expressions/routines.
- A multiple meta level interpreter.
- Interrupt-driven input.
- Automatic reaction plan generation from a description of world model, goals, and capabilities.
- Automatic, heuristic TAP test minimization.
- Automatic TAP period assignment.
- Automatic mapping of abstract world model features to sensing primitives.

We have presented implementation details of these mechanisms, and we have described guidelines for using the representations and algorithms efficiently. In particular, we have focused on the use of abstraction in domain modeling to combat both state-space explosion and related problems with planning loops and counting domains.

CHAPTER 6

THE SCHEDULER & REAL-TIME SUBSYSTEM IMPLEMENTATIONS

In this chapter we describe the prototype implementations of the Scheduler module and the Real-Time Subsystem (RTS). We provide detailed descriptions both to clarify precisely the way CIRCA is intended to operate, and to demonstrate the practicality of meeting the functional constraints imposed by the architecture.

6.1 The Scheduler

In the final phase of generating TAP control plans, the AIS sends the accumulated information about the planned 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 CIRCA diagram of Figure 1.1 showed the Scheduler as a separate entity from the AIS, communicating over explicit links. For ease of development and experimentation, those links have been simplified to simple procedure calls within the AIS; the Scheduler is currently implemented in Lisp within a KS run by the AIS interpreter. The main cost of that simplification is that the scheduling process can no longer be performed in parallel with other deliberative processing. In the Puma domain, this has little or no effect, since the scheduling process is much less time-consuming than the TAP planning.

The Scheduler uses a modified deadline-driven scheduling algorithm [42, 78] to build a TAP schedule. This algorithm specifies that, each time the system can choose which TAP to run, it should run the available TAP with the earliest deadline. To derive a cyclic schedule with this mechanism for choosing the next TAP to run, 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 that all of the TAPs that must be scheduled have been invoked at least once, 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-PERIOD`s.

If the Scheduler cannot build a schedule that guarantees all the TAP timing constraints,

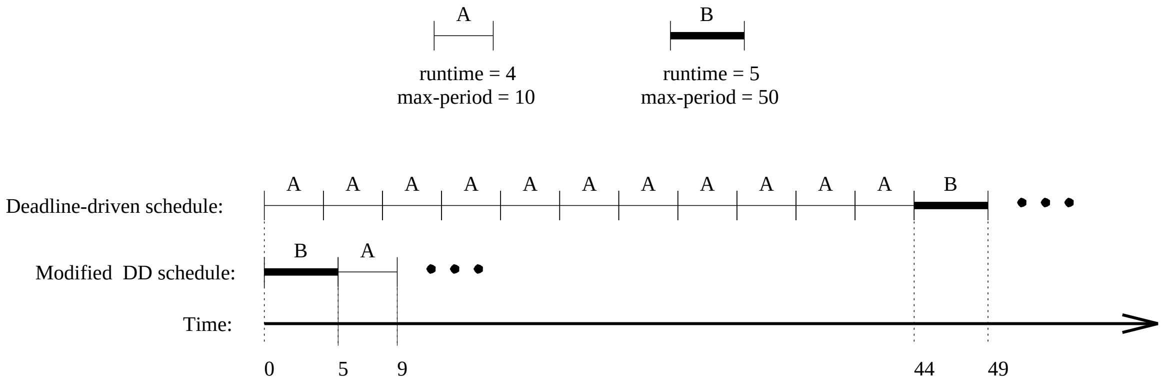


Figure 6.1: A simple example of how pure deadline-driven scheduling can produce undesirable, lengthy schedules.

it will return a failure message to the AIS. At that time, the AIS may backtrack to generate a different proposed TAP plan, or it may make other alterations to its world model to trade off some aspect of its performance, in an attempt to relax the scheduling constraints that made a TAP schedule impossible to find. Chapter 7 will discuss those tradeoff methods in detail.

6.1.1 Modified Deadline-Driven Scheduling

The simple deadline-driven criterion for selecting the next TAP to run is optimal in the sense that, if any schedule is possible, this method will produce one. However, given the scheduling problem posed to CIRCA's Scheduler, the deadline-driven algorithm does not produce particularly efficient schedules. As a simple example, consider the problem of scheduling two TAPs, A and B, where A has a runtime of 4 seconds and a maximum period of 10 seconds, and B has a runtime of 5 seconds, with a maximum period of 50 seconds. If we use the trivial deadline-driven algorithm, the schedule of TAPs will have 11 invocations of TAP A, followed by one invocation of TAP B, and then repeat that pattern, as illustrated in Figure 6.1. This schedule is perfectly acceptable because it meets the frequency requirements for both of the TAPs. However, it is clearly not the shortest schedule that meets those requirements. In fact, the very simple schedule composed of alternating invocations of A and B also meets the requirements, and it is much shorter. This short schedule length is a major advantage from the CIRCA perspective, because the Scheduler is simulating this scheduling process forward to generate the appropriate loop of TAPs. The longer the loop, the longer it takes to generate, and the more resources that computation consumes. Therefore, we have modified the basic deadline-driven algorithm so that it will tend to produce shorter, more compact schedules. The modifications do not alter the optimal nature of the Scheduler: in the worst case, the Scheduler will essentially use just deadline-driven scheduling.

The primary change to the scheduling algorithm is the addition of a second "level" of scheduling priorities, used to schedule TAPs that would not necessarily be chosen by the deadline-driven criterion, when slack time is available. Slack time is defined as the time between the current instant in the simulated schedule, and the latest possible start time of the TAP T_{DD} chosen by the deadline-driven criterion. Essentially, this slack time is the amount of time available for other processing, before the system definitely must run T_{DD}. If other TAPs can be found which fit within this slack time, they can be scheduled to run before T_{DD}. Therefore, the Scheduler first finds T_{DD}, then finds the set of "feasible" TAPs whose runtimes will fit in the consequent slack time. If none are available, T_{DD} is chosen to run next. Otherwise, to maintain a "fair" distribution of invocations of the TAPs, the Scheduler chooses to run the feasible TAP which was least-recently invoked in the schedule so far. This has the effect of producing a modified round-robin effect, rotating the privilege of a slack-time invocation among the feasible TAPs. This is not a perfectly fair round-robin, because at each scheduling point different sets of TAPs may fit within the slack time of the current T_{DD}.

With these modifications to the TAP selection criterion, the Scheduler is able to produce the second schedule shown in Figure 6.1. At time 0, the simple deadline-driven criterion indicates that T_{DD} is TAP A, because its deadline is 10, while TAP B has a deadline of 50. However, because A's runtime is 4, there are 6 seconds of slack time before it must be invoked. Since B's runtime of 5 fits in that slack time, the system selects B to run first. At the next scheduling point, time 5, T_{DD} is A again, but this time only 1 second of slack time remains, B will not fit, and A is selected[1]. At this point, since both A and B have been scheduled, the system will begin scanning for acceptable loops in the schedule so far, and the simple loop BA meets all constraints.

As an example of how the Scheduler can fail, consider what would happen if TAP B had a runtime of 7 seconds. At time 0, the deadline-driven criterion would select TAP A, and now B would not fit in the slack time, so A would be scheduled. This would continue as shown in the upper schedule of Figure 6.1 until time 44, at which time B would be selected. However, at this point the invocation of B would finish after its deadline of 50, and the Scheduler would recognize this error condition.

Actually, this simple example can easily be recognized as unschedulable without any forward simulation. If we consider any single invocation of TAP A, we can see from its period and runtime that there will be at most $10 - 4 = 6$ seconds of time available for other TAPs between the required invocations of A. Thus the seven-second runtime of B in this example makes it impossible to ever schedule these two TAPs together. More generally, it must always be that case that, for any two of the N TAPs being scheduled, the sum of their runtimes (worst-case execution times) is less than either TAP's period.

6.1.2 The If-time Server TAP

If the Scheduler is able to produce a TAP schedule that includes all of the TAPs that must be guaranteed, it is possible that there are enough slack resources in the RTS to also guarantee some of the if-time TAPs. Putting if-time TAPs into the guaranteed schedule can have the beneficial effect of speeding CIRCA's reactions.

[1] Actually, even if B did fit it would not be selected here; the T_{DD} TAP is included in the least-recently-run round-robin, so A would be selected.

One way to achieve this benefit would be to iteratively include additional if-time TAPs in the list of TAPs being scheduled, increasing the number until the Scheduler finally fails. At that time, the last successful schedule could be retrieved, and it would provide some of the if-time TAPs with guaranteed, scheduled invocations. The main problem with this simple iterative approach is that it does not share the benefits of the slack time evenly over the if-time TAPs: it is not "fair." Whichever if-time TAPs actually get scheduled receive the full benefit of being guaranteed, while the remaining if-time TAPs may never be invoked at all, because they remain on the if-time list.

To avoid this difficulty while still taking advantage of possible slack resources, we have implemented an "if-time server" TAP, which tries to fairly distribute the available slack time amongst all of the if-time TAPs. Instead of scheduling individual if-time TAPs when slack resources are available, the AIS builds an instance of the if-time server TAP and passes it to the Scheduler with the guaranteed TAPs. When executed by the RTS, the if-time server TAP performs its own round-robin over the if-time TAPs. On each invocation, the server TAP executes the if-time TAP pointed to by its round-robin pointer, and then increments that pointer to the next if-time TAP. The overhead of the server TAP is extremely small, because it only increments that single pointer.

The if-time server TAP could also implement a more complex method of choosing the next TAP to run. For example, the server could use priorities assigned to TAPs, and maintain a multi-level priority queue similar to an operating system. Or, the server TAP could be given additional knowledge of the domain and the constraints between TAPs, and select appropriate TAPs to run based on that information. There are two constraints on this sort of more complex server. First, the overhead of the server would be increased by this complexity, consuming more of the available slack time. Second, the additional knowledge of priorities or other information must be derived and built into the server TAP. For now, the round-robin server provides a low-overhead, low-information alternative that distributes slack time as evenly as possible.

The server TAP is able to invoke any of the if-time TAPs because the AIS builds it with a worst-case execution time set to the maximum of the worst-case execution times of all of the if-time TAPs. Deciding on a `MAX-PERIOD` to assign to the server TAP is somewhat more difficult. Ideally, the if-time server TAP would be given a period that would cause it to be invoked frequently enough to use the slack resources, but not so frequently as to make a schedule impossible. Since it is not possible to directly determine this value, we have implemented a simple iterative heuristic to try to optimize the `MAX-PERIOD` assigned to the if-time server TAP.

As a starting point, the server TAP is assigned a `MAX-PERIOD` equal to the period of the first schedule produced (containing just the required TAPs, without any if-time server TAP). Thus if all other scheduling constraints are met, the server TAP will be invoked once per cycle through the new TAP schedule. If the Scheduler is able to produce a successful schedule with these constraints, the AIS then decreases the server's period by some amount (currently by 25%), and repeats the scheduling process. This iteration terminates when the Scheduler fails, and the last successful schedule is restored and used. If the initial server TAP `MAX-PERIOD` assignment does not result in a feasible schedule, the AIS can also increase the value iteratively for a few cycles, until a successful schedule is produced.

6.1.3 Discussion

We have shown how the Scheduler is able to produce cyclic schedules of TAPs that can be shorter than those built by simple deadline-driven scheduling, and how the if-time server TAP can allow the system to make guaranteed utilization of slack time. It is important to note that our modifications to the deadline-driven algorithm take effect only when the schedule utilization is fairly low; when the utilization is high, slack time is minimized and the second level of scheduling is never possible. As a result, our modifications alter the Scheduler performance for low-utilization domains, but in worst-case, high-utilization domains they have no effect, and the system defaults to pure deadline-driven scheduling.

Experiments using these mechanisms on hundreds of variations of the Puma domain (involving different goals, part arrival rates, emergency alert rates, etc.) have shown that the Scheduler produces efficient, short schedules very quickly, or else rapidly recognizes that a particular set of TAPs is not schedulable. Most Puma domain schedules consist of between 15 and 35 TAP invocations, and are generated in well under a minute.

However, in the worst case, the Scheduler might have to construct a schedule as long as the maximum possible loop size, equal to the least common multiple (LCM) of the TAP `MAX-PERIOD`s. While this does not pose a problem for many hand-crafted real-time systems, in which the task periods are carefully arranged to be simple multiples of each other, the automatically-generated TAP periods created by CIRCA are not so convenient. For example, in one version of the Puma domain, the maximum possible schedule loop for ten TAPs includes at least 10^{42} TAP invocations[2]. Thus the implementation of the Scheduler within an interruptible KS is a good choice; the AIS can use a timer interrupt to make sure that the Scheduler returns a result within a reasonable amount time, as described in Section 5.2.1. If the timer interrupt halts the Scheduler, then the AIS might decide that some modifications are necessary to the planned TAPs to make the scheduling processing easier.

For example, the AIS might decide to use a heuristic method for reducing some of the TAP periods so that they have a smaller LCM, thus making the worst-case schedule much shorter. By only reducing periods, not increasing them, this sort of modification retains or improves upon the response-time guarantees that motivated the original period assignments. However, while the worst-case schedule may be shorter, the schedulability of the set of TAPs is also decreased; shorter TAP periods mean higher utilization, making it more difficult to fit all the TAPs into a schedule. Furthermore, there is no obvious heuristic for choosing how much to decrease the TAP periods to achieve a useful LCM. Currently, this modification has not been implemented.

[2]As it turns out, the successful schedule loop for that example required only 21 TAP invocations.

6.2 The Real-Time Subsystem (RTS)

The RTS was originally prototyped in Lisp, and has since been re-implemented and enhanced in C. Both versions have the same basic functionality, but the C version provides increased speed, efficiency, and predictability.

The main program loop of the RTS is shown in C-like pseudo code in Figure 6.2. The RTS begins by initializing numerous variables and communication links and then loading a bootstrap TAP schedule. The bootstrap schedule is designed to read in a new TAP schedule as soon as possible from the AIS. After that initialization, the RTS simply executes the current TAP schedule as long as the **run-current-schedule** flag is set. If that flag is turned off by a TAP, that indicates to the RTS that a new schedule has been read in, and the RTS will drop out of its TAP-execution loop just long enough to install the new schedule. Switching to a new schedule is merely a matter of adjusting several pointers, such as the pointer indicating the current TAP within the schedule being executed.

Within the TAP-execution loop, the RTS runs through the guaranteed TAP schedule, evaluating the test expression for each TAP and firing those TAPs whose tests return true. If a guaranteed TAP does not use all of its allocated worst-case execution time, the RTS uses the resulting slack time to search for and invoke one or more of the unguaranteed, if-time TAPs. The decision to look for additional if-time TAPs to execute takes into account the overhead of the RTS processing time itself.

The pseudo-code of Figure 6.2 is simplified in several ways from the actual C code. One major difference is that the TAPs for the current schedule are not kept in a linked list, as implied in the pseudo-code. Instead, the RTS is built with two large pre-allocated arrays of TAP structures. At any one time, one of those arrays holds the TAPs currently being executed and the other array may be loaded with the TAPs for the next schedule. The **switch-to-new-schedule()** routine simply swaps the array pointers used by the TAP execution and TAP downloading routines.

The primary motivation for this TAP array mechanism is to avoid replicating the TAP structures. A single TAP schedule may contain many invocations of each TAP, particularly when the TAP periods are diverse. If the TAPs were stored as a linked list (as they were in the Lisp-based RTS), then each invocation of a TAP would correspond to a replication of the TAP structure. This would be very inefficient, since the data within the replicated structures would be completely identical, except for the pointer to the next TAP. The array system avoids this problem, because TAPs are represented independently from the TAP schedule; each TAP is only stored once. Another pre-allocated array is used to hold the schedule, which is now represented as simply a list of indices into the TAP array. Figure 6.3 illustrates the actual details of the implemented storage scheme. The array mechanism avoids a potentially large amount of data replication that might result from an inefficient method of storing TAPs. In the earlier version, this data replication not only used storage space inefficiently, it also slowed down the communication of TAP schedules to the RTS; each TAP in the schedule was transmitted to the RTS in order, so the replication was propagated over the communication channel as well. The use of pre-allocated arrays and schedules represented by array indices avoids this inefficiency, and also avoids the time cost of allocating a new TAP structure each time a TAP is downloaded.

```
initialize_rts();
load_bootstrap_schedule();
while (!halt)
  {
    while (run_current_schedule)
      {
        start_tap_time = current_time();
        if (execute_test_expression(cur_tap)) execute_action(cur_tap);
        cur_tap = cur_tap->next;
        end_tap_time = current_time();
        slack_time = cur_tap->wcet - (end_tap_time - start_tap_time);
        while (slack_time > rts_iftime_overhead)
          {
            start_tap_time = current_time();
            if ( cur_iftime_tap->wcet < slack_time - rts_iftime_overhead
                 && execute_test_expression(cur_iftime_tap) )
               execute_action(cur_iftime_tap);
            cur_iftime_tap = cur_iftime_tap->next;
            end_tap_time = current_time();
            slack_time -= end_tap_time - start_tap_time;
          }
      }
    switch_to_new_schedule();
  }
```

Figure 6.2: Pseudo-code for the RTS main loop.

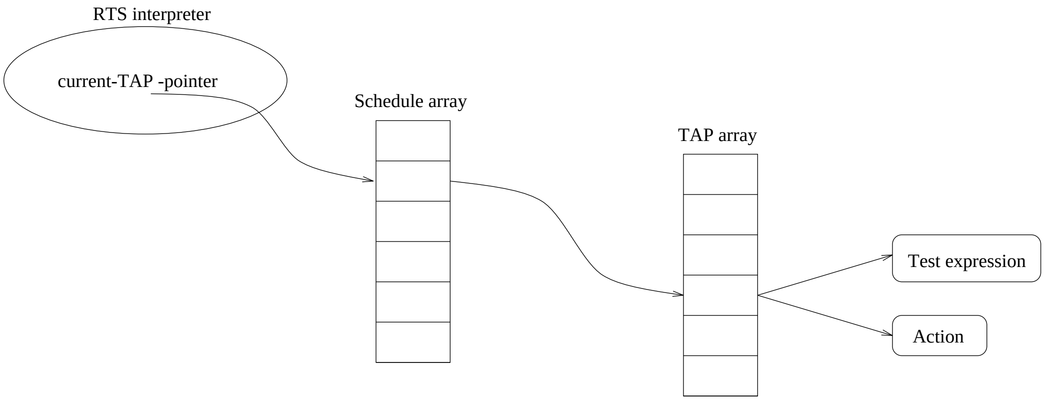


Figure 6.3: The array-based storage scheme for TAP schedules.

```
char buffer[BUFFER_LENGTH];
int buffer_position;

void get_new_schedule()
  {
     int read_length, end_of_message;

     read_length = read_socket_message(AIS_socket,buffer+buffer_position,
                                        MAX_READ_LENGTH, &end_of_message);
     buffer_position += read_length;
     if (end_of_message)
       {
          parse_new_schedule(buffer);
          buffer_position = 0;
          new_schedule_ready = 1;
       }
  }
```

Figure 6.4: The **get-new-schedule** function.

6.2.1 Downloading a New TAP Schedule

From just the main RTS code loop of Figure 6.2, it is not clear how the RTS ever gets a new TAP schedule. The approach is simple: the TAP schedule itself includes a TAP that executes the **get-new-schedule** function, illustrated in Figure 6.4. This function causes the RTS to read in a new TAP schedule from the AIS, as part of the processing of the current schedule. Therefore, by building this TAP into the schedule, the AIS actually determines how often the RTS is checking to see if a new schedule is available. When the AIS expects the environment to be highly dynamic and challenging for the RTS, it can cause the RTS to spend less time checking its input communication buffers, by increasing the maximum period of the **get-new-schedule** TAP.

The **get-new-schedule** function is crucial to the guaranteed performance of the RTS because it reads in the new TAP schedule incrementally. Each time there is data waiting from the AIS, the RTS will read in a constant amount (`MAX_READ_LENGTH`) of the new schedule from the AIS. This has the effect of interleaving the downloading of the next schedule with the execution of the current schedule, avoiding unpredictably long periods in which the RTS is involved in communication. All incoming communication is broken up into fixed-size packets whose processing is explicitly scheduled.

When the AIS has sent a complete new TAP schedule, it terminates the message to the RTS with a special flag character ('#'). The **read-socket-message()** function detects this character and sets the **end-of-message** flag, indicating that the schedule has been completely read into the RTS input buffer. At that time, as shown in Figure 6.4, the RTS parses the new TAP schedule all at once. This non-incremental behavior is not desirable, but proved easiest to implement with available automatic parser generators. Extensions to

```
TAP switch-to-new-schedule
   :TEST  (schedule_ready T)
   :ACTION  (run_new_schedule)
   :MAX-PERIOD 0                          ;; An if-time TAP.
   :TEST-TIME .0005
   :ACTION-TIME .005
```

Figure 6.5: A simple TAP used to transfer control to a new TAP schedule.

more powerful, incremental parsers should be completely straightforward. The grammar of the TAP schedule download language is described in Appendix B, which also includes an example schedule download message. Once the new TAP schedule is completely downloaded and processed, the global **new-schedule-ready** flag is set.

6.2.2 Transferring Control to a New TAP Schedule

Switching control to the new TAP schedule is somewhat complex, because the system must continue to ensure its safety during and after the switch. From the graph model viewpoint, the system must only switch to a new schedule when the world is in a state shared by the models used to generate both the old and new schedules, as discussed in Section 4.6. The world states accounted for by the new schedule must include at least one state that is also reachable with the old schedule. If the new schedule completely subsumes the old, then the switch can occur at any time.

This is most often the case in the Puma domain, where the AIS is usually simply adding more capabilities to the TAP schedules by deriving new part-packing methods. As a result, the new TAP schedules can handle all of the same world states as the old schedules, and more. For these simple transfers, the TAP illustrated in Figure 6.5 is sufficient to accomplish a switch to the new schedule, using the RTS primitive functions illustrated in Figure 6.6. The TAP simply tests to see if a new schedule has been completely received, and, if one has, it resets the **run-current-schedule** flag. The RTS then terminates the TAP execution loop, performs several pointer swaps, and immediately returns to executing the new TAP schedule, as shown in Figure 6.2. Thus the transition between TAP schedules is extremely rapid, and can be subsumed by the execution time requirements of the **switch-to-new-schedule** TAP without great cost.

In other domains, it is more likely that the AIS will have to decide on one or more states from which it will switch to the next plan phase, and download TAPs to detect when the world is in one of those states and a new TAP schedule is available. For example, in a mobile robot domain each TAP schedule might be used to implement a distinct phase of a path plan, and the transfer of control between plans should only be accomplished when the robot has reached the end of one TAP plan's path region. The world states possible once the robot has reached the end of the first TAP plan's path are presumably shared with the TAP plan for the next path region. In that case, the transfer TAP's test expression might be modified to read something like: **(and (robot-status at-destination) (schedule-ready**

```
int schedule_ready ()
  {
    if (new_schedule_ready) return(T);
    else return(NIL);
  }

void run_new_schedule ()
  {
     run_current_schedule = 0;
  }
```

Figure 6.6: RTS primitives used in transferring control to a new TAP schedule.

T)).

When building a TAP plan P_i, it is possible that the AIS will not be able to decide ahead of time which world model states are appropriate for the transfer of control to the subsequent TAP plan P_j. In that case, the AIS can still implement a safe transfer by using a combination of the methods described above. First, the AIS builds into the current TAP plan P_i the simple **switch-to-new-schedule** TAP, so that P_i will transfer control to a new schedule as soon as it is downloaded, with no other conditions required. Then, once the AIS has derived the subsequent TAP plan P_j and the appropriate states in which a transfer should be made from P_i to P_j, the AIS downloads to the RTS a slightly-modified copy of P_i, in which the trivial **switch-to-new-schedule** TAP is replaced by a TAP that transfers control only when it detects the appropriate states[3]. The RTS can swap in this new plan without risk, because the new plan completely subsumes the current plan P_i. Finally, the AIS can download the new plan P_j, and the transfer will be accomplished in the appropriate states.

6.2.3 Feedback to the AIS

Communication out of the RTS to the AIS is also accomplished only within TAPs, as illustrated in Figure 6.7. The **unknown-part-arrived** TAP detects when unknown-shaped parts arrive, and notifies the AIS that a new part-packing plan is required. The TAP includes an explicit message-sending function, **notify-AIS**, whose execution time is included in the `ACTION-TIME` for the TAP. Similarly, the I/O time required to communicate with the sensors and actuators is included in the timing characteristics of the TAPs that use those channels. Thus all communication into and out of the RTS is scheduled explicitly within TAPs, avoiding unpredictable I/O delays.

This scheduled communication not only allows the RTS to behave predictably, it also gives the AIS control over the amount of feedback data which the RTS sends to the AIS, allowing a dynamic filtering similar to that used by Guardian [28]. If the AIS needs to keep

[3] This more-complex test expression will increase the transfer TAP's worst-case execution time. Either the original **switch-to-new-schedule** TAP can be scheduled with extra time, or the schedule can be re-generated with the new parameters, or the transfer TAP can be made an if-time TAP.

```
TAP unknown-part-arrived
   :TEST  (type_of_conveyor_part unknown)
   :ACTION  (notify_AIS unknown-arrived)
   :MAX-PERIOD 0                            ;; An if-time TAP.
   :TEST-TIME .0025
   :ACTION-TIME .04
```

Figure 6.7: A feedback TAP that detects when unknown-shaped parts arrive and notifies the AIS.

close track of an environmental feature, such as the state of the Puma-domain emergency alert light, or the charge-status of a mobile robot's battery, it can build a TAP that will send the value of that feature back to the AIS as frequently as necessary. Or, if the AIS only needs to be notified of rare failures or undesirable events, it can plan less-frequent feedback TAPs. If some feedback information is optional, the AIS could plan if-time TAPs to send the data only if the RTS is not busy performing other, more important tasks.

There are two significant research issues related to such feedback TAPs:

- How does the AIS decide which world model states to select for feedback TAPs?
- How does the AIS decide what information the feedback TAPs should return to the AIS?

Neither of these issues has yet been fully addressed. However, we have begun investigating preliminary approaches to these problems.

When to Send Feedback

Feedback TAPs appear to be useful in two general situations. First, the AIS might want to be notified of any type of progress in the system, to ensure that the TAP plans are achieving their goals. For example, the AIS might like to be notified when each new TAP plan takes control, so that it can monitor the progress of the plans. The need for this type of feedback is outside the limits of the world model used for TAP planning: the world model does not represent the state of the AIS' knowledge. Since our focus has been on planning TAPs using the world model, we have not investigated general progress-monitoring feedback TAPs.

We distinguish this general need for progress reports from the second type of feedback, used to report that a TAP plan has encountered some problem which might prevent it from achieving all its goals. For example, when a part of unknown shape arrives, the RTS must put it on the table, and it cannot pack that part into the box until a new TAP plan is provided. This means that the RTS will not be able to achieve its **(part-on-table nil)** goal. In terms of the world model used for TAP planning, the system has reached a *dead-end* state. That is, the world has now entered a state from which there is no way to reach any ideal state achieving all the system's goals. Note that this does not mean a control-level goal will be violated: if the system has guaranteed all of its control-level goals, there is no risk of catastrophe. Only the achievement of task-level goals can become jeopardized in this way. This characteristic can be used by the AIS to automatically recognize situations in which feedback TAPs should be executed by the RTS to alert the AIS.

We have implemented a preliminary module for detecting dead-end states in the world model and building feedback TAPs for them. The current version successfully locates the many dead-ends in the Puma domain (including all the states in which an unknown-shaped part has already been placed on the table). In practice, the only problem with this approach is that it results in too many notification messages to the AIS: after an unknown part has arrived, *every* world state is a dead-end.

We are beginning to investigate two approaches to solving this problem. The simplest approach is to make the feedback TAP "one-shot," so that it sends a single message to the AIS and then disables itself. This is a practical, simple approach but clearly not ideal, particularly because it is outside the scope of the world model's representation.

The second approach is to detect the move into the dead-end region, notifying the AIS on that transition and not otherwise. This approach shows promise for limiting the number of feedback messages to the AIS and keeping the representation and motivation for that limitation within the bounds of the world model. This approach might lead to a TAP like the one illustrated in Figure 6.7, detecting when the unknown-shaped part first arrives, because after that the current TAP plan can never maintain all of its goals.

What Feedback to Send

A significant research issue that has not yet been addressed is how the system decides exactly what information should be returned to the AIS. For example, in the Puma-domain example above, it is clear that the AIS needs to know more than simply that an unknown-shaped part has arrived; the AIS also needs to know something about the shape of the new part. One possible solution is to have the RTS send back to the AIS all the information it has about the domain, including the shape information it extracts from a camera image or other sensor modality. This approach is certain to provide all the necessary information to the AIS, but it is tremendously inefficient. The RTS may have a great deal of sensor data available, and one of its purposes is to isolate the AIS from that complexity.

An alternative approach would be to have the AIS reason about what features distinguish the feedback-triggering world model state from the similar states in which feedback is not required. In the example scenario, the AIS has planned a feedback action for the arrival of unknown parts, but the similar arrival of known parts is dealt with by other planned actions. Thus a comparison of the respective world model state descriptions would reveal that the feedback-trigger state differs in the **(type-of-conveyor-part unknown)** feature. The AIS must then associate that feature with the related sensor information and decide what should be sent as feedback.

6.3 Discussion: The RTS Really is Real-Time

We have stressed that CIRCA combines the ability to run arbitrarily complex, unpredictable AI methods with guaranteed, predictably real-time performance of critical control tasks. It is trivial to prove that CIRCA's AIS is capable of implementing very complex algorithms: the system is clearly Turing complete, since it incorporates arbitrary Lisp code. Therefore, to completely justify our claim of combined real-time and AI, we must show that the RTS really can provide predictable, guaranteed real-time performance, and that the RTS guarantees are truly isolated from the AIS.

This seems an appropriate time to reiterate the fact that the goal of real-time systems is not to be "fast," but to be "predictably fast enough." That is, a real-time system must be known to operate at a rate sufficient to meet the demands of its environment. Since mere processing speed is easily varied by using different computer hardware, the most important aspect of real-time systems is actually predictability.

To show that CIRCA's RTS is truly predictable, and thus that it provides a suitable execution environment for TAPs implementing real-time reactions, we will examine the possible sources of unpredictability in the system, including communication delays, context switching, and dynamic memory. We will describe how the RTS avoids unpredictability in dealing with each of these potential problems.

6.3.1 Communication and Interrupts

From an architectural standpoint, the I/O mechanisms of the RTS are perhaps its most important features, because they provide the crucial isolation of the predictable performance of the RTS from the uncertain operations of the AIS and Scheduler. Unbounded communication delays are avoided by making all socket I/O calls nonblocking using fixed-maximum-length operations. Calls to **socket-read**, for example, are made with a limited input buffer size to be filled, and they return immediately no matter how much (or how little) data is available on the connection. Calls to **socket-write** return immediately after putting a limited amount of data onto the connection, whether or not the receiving end has gotten that data yet. Therefore, given a real-time operating system with well-understood system calls having bounded behavior themselves, the communication in and out of the RTS is incapable of causing unexpected delays, and the RTS remains fully predictable even while communicating with the AIS.

Unlike many systems which attempt to ensure real-time performance through rapid response to interrupts, the RTS does not accept any interrupts. As discussed above, the RTS is expected to have TAPs that explicitly check for all important conditions as quickly as necessary. In a sense, we have moved the polling loop out of the interrupt hardware and into the software RTS, so that the AIS and Scheduler can reason explicitly about the form and frequency of that loop. Moving the polling loop decreases its frequency, since many processor instructions are involved in running each TAP. For example, the pSOS$^+$ real-time kernel can provide interrupt service in 6 microseconds (see Section 6.3.7), while the fastest possible TAP schedule can only respond in about 70 microseconds (see Section 6.4)[4].

[4]Interestingly, when running under Unix, the response time using polling in the RTS is much faster

However, moving the polling loop to software increases the architecture's ability to control and predict the responses of the system. If the RTS accepted interrupts, it would be very difficult to make any guarantees about its performance, since the system would have to account for the many unpredictable aspects of interrupt-driven systems. For example, lower-priority interrupt handlers could never be guaranteed, since higher-priority interrupts would preempt and override their behaviors. Furthermore, incoming interrupts might be lost if they arrived during the handling of equal- or higher-priority interrupts. Because of these and other problems, interrupt-driven systems are less suited to predictability and guarantees than the polling behavior of the RTS.

6.3.2 Dynamic Memory Allocation

Because new TAP schedules are downloaded from the AIS and are not known *a priori*, the RTS must have the ability to dynamically allocate storage for the new TAP structures and schedules. This allocation is performed within the **get-new-schedule** TAP that reads in and parses a new schedule, and it is therefore fully scheduled. The maximum amount of memory that may have to be allocated in a single **get-new-schedule** TAP invocation is determined by the maximum size message that the TAP can read in from the AIS. Naturally, the host computer's operating system must provide a predictable system call to implement the allocation, and the system must be provided with enough memory for the task. If a suitably guaranteed operating system primitive is not available then the RTS could be implemented to preallocate a large amount of memory before bootstrapping, and then allocate that memory itself in a bounded, predictable manner.

To avoid running out of memory for allocation, the RTS should also deallocate memory that was allocated for TAPs used by schedules that are no longer being run. There are several ways to implement this functionality, in order to spread the cost of deallocation in different ways. If the cost of deallocation is sufficiently small, the code which reads in a new schedule can deallocate the memory allocated to the TAP schedule which was last running (not the one that is currently running). In the current implementation, this method is available but is not necessary, because the TAP schedules we have investigated are not large enough to tax the several megabytes of available memory; even the larger Puma domain schedules use less than 2500 bytes of memory per schedule, so the AIS would have to send at least 400 new schedules to use up a single megabyte of RTS memory. However, memory preservation could prove crucial for larger-scale TAP schedules, and distributing the cost of deallocation across the multiple invocations of the **get-new-schedule** TAP may be useful.

An alternative would be to avoid deallocation completely unless the AIS deems it necessary. The AIS could either model and keep track of the amount of memory the downloaded TAP schedules will have consumed, or it could have the RTS run a TAP specifically designed to watch for low-memory conditions and notify the Scheduler and AIS if such a situation arises. The AIS would then download TAPs which would include memory deallocation operations (and the Scheduler would be sure of not building schedules larger than the available

---

than using interrupts, because of the expensive Unix context switch. The interrupt response time on the Unix-based RTS host machine was measured at 25000 to 90000 microseconds!

RTS resources). The advantage of this approach is that it allows the system to avoid all of the overhead of deallocation when the overhead is not absolutely necessary. A disadvantage of this approach is that, because deallocation is put off as long as possible, it may lead to temporarily unacceptable performance degradation when the system must suddenly spend much of its time deallocating and reorganizing memory.

In many ways, these approaches parallel the methods used for dynamic memory and garbage collection in Lisp. The main difference is that the RTS does not have a difficult task in discovering which memory elements are no longer used: any allocated memory not being used by the current TAP schedule (or the schedule currently being downloaded) is unused, and may be deallocated. As a result, incremental deallocation is actually quite simple and fast for the RTS. In fact, since the pointers to the last-run schedule are available while a new schedule is being read in, it is trivial to recognize and deallocate those memory locations before allocating new memory for the new schedule. However, if we put off the deallocation indefinitely, the RTS will have to implement some new functionality to keep track of old schedules and their memory allocations, for later deallocation on demand. Thus incremental deallocation is much simpler for the RTS, and has the advantage of spreading the overhead more smoothly over the system's operations.

TAPs themselves may perform dynamic allocation of memory, if that allocation is considered as a resource consumption by the Scheduler, and thus is known to cause no problems. In the current implementation, TAPs could perform allocation because the C primitives they invoke have that ability, but we have found no use for this mechanism (yet). In the Puma domain, for example, proper use of indexical variables avoids the need to "gensym" a new symbol or variable for each instance of a block (as discussed more fully in Section 5.4.5).

6.3.3 Context Switching

Once a new TAP schedule is input to the RTS, the system must perform a context switching operation to begin executing the new schedule. As described in Section 6.2, the RTS implements this capability in an extremely efficient and completely predictable manner. The context switch is accomplished by a TAP action which sets the flag **run-current-schedule** to **FALSE**, making the inner RTS loop terminate. Falling out of that loop, the RTS performs several pointer swaps to make the relevant array variables point to the new schedule arrays (guaranteed TAPs, if-time TAPs, and schedule of TAPs), and a series of variable initializations to make the RTS read the next incoming schedule into the unused arrays. Thus the context switch does not require any looping or other complex computations, and has bounded time and resource requirements. In fact, these resource requirements are included within the specification of the RTS primitive which triggers the context switch, so that even the time used outside of the TAP loop, in the context switch code, is predicted and scheduled.

6.3.4 Shared Resources

Contention over shared resources is another potential source of unbounded delays. However, this poses no difficulty for the RTS because all resource usage is scheduled within

```
ACTION push-emergency-button
      PRECONDS: ((robot-status free) (part-in-gripper nil))
      POSTCONDS: ((emergency nil) (robot-position over-button))
      RESOURCES: (arm gripper)
      WCET: 3.5 [seconds]
```

Figure 6.8: The **push-emergency-button** action transition.

TAPs (including the context switch time noted above). TAPs which require resources other than time (such as a TAP that requires the Puma gripper) will automatically include tests to make sure that those resources are available, unless the AIS has determined that these tests are redundant and unnecessary, based on its simulation of the possible world states (see Section 5.3.2).

For example, consider the simple **push-emergency-button** action transition description shown in Figure 6.8. The preconditions specify that the robot must be free (i.e., not busy) and its gripper must be empty. After the planner determines the 54 states for which this action is appropriate, the resulting TAP has the simple tests: **(and (part-in-gripper nil) (emergency T))**. Note that the TAP does not bother testing the **robot-status** feature, despite the fact that the robot (a resource) is actually required for the TAP's action. The test generalization phase has determined that this test is unnecessary, because any time this TAP is executed and the gripper is not holding anything, then emergency alerts are the most important priority– the robot will not choose to do anything except respond to the emergency. Contention for resources is automatically avoided, because only one action is planned for each possible world state.

6.3.5 Additional Sources of Uncertainty

If the RTS allows TAPs to execute loops or recursive functions, it must be sure that those program structures will still have a predictable worst-case performance. Therefore, although the system does not restrict the structure of RTS primitives, the user is required to specify a worst-case processing time for each primitive. If looping or recursion is involved, the user may ensure bounded resource usage either by using any-time algorithm methods (see Section 2.2.2) or by otherwise limiting the worst-case number of iterations or recursions that will occur. This does not preclude programming structures whose termination condition is determined at run-time; a worst-case bound must always be available, but the primitives are free to use less than that amount of time.

Faults in processing hardware, software, I/O devices, sensors, or actuators might lead to unpredictable performance by the RTS and the devices it controls. The RTS does not yet make any provisions to deal with such faults: it is currently assumed that lower-level fault-tolerant mechanisms are available to detect and correct all possible faults. However, the design of CIRCA has been tailored for future extensions in which fault-tolerance issues can be addressed through the use of "homeostatic" [4] or internal-state monitoring and control primitives. The RTS could execute TAPs which examine the state of its execution environment to detect all types of faults and implement short-term, control-level workarounds for intermittent or temporary system failures. Longer-term faults would trigger feedback communication to the AIS and Scheduler, which would then modify their models of the capabilities of the RTS and the system it controls, to represent the faulted component. With this modified world model, the AIS and Scheduler would then automatically build new control plans that account for the faults in the system, and CIRCA as a whole would be cognizant of its faults and able to make rigorous statements about its behavior despite those faults. Of course, the immediate real-time response to faults would remain the responsibility of the RTS: the system's behavior immediately following a fault would only be guaranteed if the AIS had predicted the problem and scheduled a TAP to at least prevent any potential control-level failures.

For example, suppose that one of a mobile robot's obstacle-detecting sensors fails, and the RTS is able to detect this failure during the execution of a monitoring TAP. The TAP could then send a message to the AIS indicating that the sensor is permanently unavailable. Furthermore, because the AIS has predicted that this problem might occur, the RTS will have TAPs that switch to an alternate sensor or perhaps implement some other actions (such as halting the robot) that will prevent the system from failing by colliding with obstacles. When the AIS builds the next control plan, it will never try to invoke the failed sensor because of the modified world model, so the fault will be taken into account. The AIS will still build plans as usual, and will still attempt to make performance guarantees, given its (now more-restricted) resources.

The usefulness of this type of internal monitoring and feedback is one of the motivations for keeping the CIRCA Scheduler module a separate entity from the AIS, so that it may be tied more closely to the RTS than in the current implementation. Because the Scheduler reasons about resources available to the RTS in building TAP schedules, it is a good location for detailed information about faulted system components. That information may have a direct effect on scheduling TAPs, as well as planning and building them.

As with all fault-tolerant systems, this approach would still be subject to overload and failure in the presence of an excessive number of faults, or faults in unprotected system components. In a sense, this approach to building fault-tolerant control plans is no different than the normal CIRCA planning process, which considers that various occurrences in the world may lead to unacceptable failure. The basic CIRCA planner derives TAPs that detect undesirable situations where failure is impending, and prevents the failure from occurring. Fault recognition and handling are essentially the same problem, so CIRCA's methods for planning and recognizing its ability to handle different domain deadlines are equally applicable to recognizing its ability to handle different faults.

6.3.6 The Real Problem: Unix

As noted earlier, the current RTS implementation runs under Unix, and therefore is subject to the vagaries of that operating system's scheduler. The complexity of the Unix scheduling scheme, as well as the large number of servers and other processes on a Unix multiprocessing system, make it essentially impossible to guarantee any service rate or computation speed for a given task. Therefore, running under Unix, the RTS cannot actually

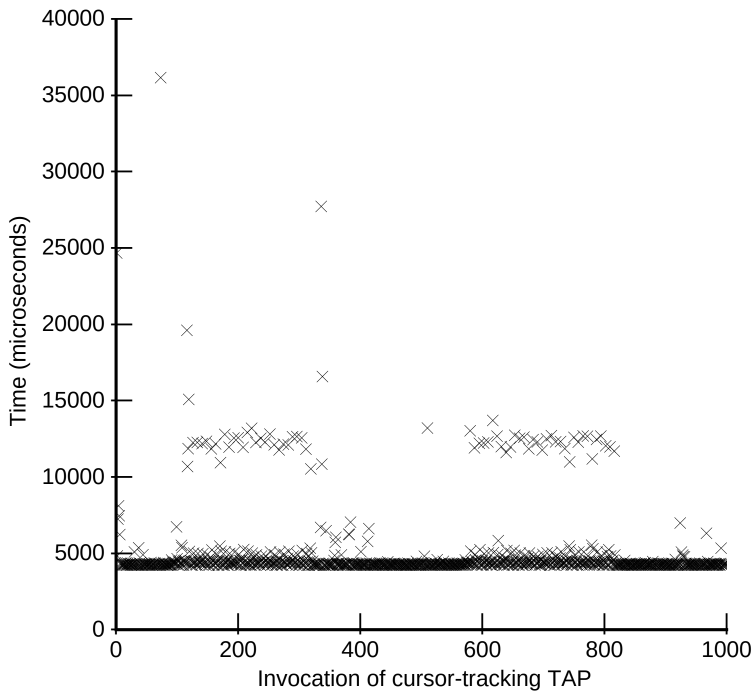


Figure 6.9: Timing behavior of a cursor-tracking TAP, showing the unpredictability introduced by Unix.

enforce real-time guarantees in a rigorous fashion.

To illustrate the problem, Figure 6.9 show the timing behavior of a simple cursor-tracking TAP used in an Xwindows-based demonstration— the TAP checks to see if the cursor has moved in the demo window, and if it has it redraws a circle around the cursor. The graph shows two primary bands of timing values, one near 5000 microseconds (when the cursor did not move) and two bursts of timing values near 12000 microseconds, corresponding to two periods of cursor movement when the TAP's action was executed, using an additional 7000 microseconds to draw the circle. Interspersed among these clear characteristics are several outlying timing values that result from spurious Unix interrupts and delays. For example, the timing values over 15000 microseconds near the 100th invocation are the result of high-priority disk accesses that were audibly occurring during this test.

After briefly reviewing the steps taken to avoid some of the problems of Unix, we will discuss the potential for porting the current RTS to the pSOS$^+$ real-time executive kernel.

Ideally, the RTS would be executed on a dedicated processor which would not be shared with other tasks, thereby making the service rate for the RTS process constant. Unix allows multiple processes to share a singe processor using a flexible priority-based scheduling algorithm, so we execute the RTS with the highest possible user-task priority, in an effort to make sure that it is scheduled to run as frequently as possible. Still, system tasks such as disk and network I/O take precedence, and the RTS may be interrupted and idled for potentially lengthy amounts of time (on the order of milliseconds) when these I/O demands are high. If the Unix machine is heavily loaded, it is also possible that the interrupted RTS task could be swapped out of main memory onto disk– this would result in extremely long delays when resuming execution of the RTS. While this event is highly unlikely on the current workstations with 16 or more megabytes of memory[5], the RTS still takes the precaution of telling the Unix kernel on startup to avoid this "swapping out" behavior.

6.3.7 Porting the RTS to pSOS$^+$

The real solution to these problems with Unix would be to port the RTS to a platform running a real-time operating system, which could provide guaranteed CPU allocations for the RTS. During the main development phase of the current CIRCA prototype, we did not have access to any suitable computing platforms that were set up to control real-world or simulated devices. Therefore, we have not actually implemented the RTS on a real-time kernel. However, we have analyzed the feasibility of that task by comparing the programming features of the pSOS$^+$ real-time executive [74] with the requirements of the current RTS implementation.

The pSOS$^+$ kernel provides a fairly complete set of operating system primitives which all have completely predictable, bounded execution time. In addition, the kernel provides very fast and predictable context switching and interrupt response times. Running on a Motorola 68020 at 25MHz, pSOS$^+$ requires 6 microseconds to respond to an interrupt, and 19 microseconds to switch contexts to a new task. The current RTS design does not adhere to the philosophical orientation of pSOS$^+$, which advocates building real-time applications based largely on separate, interrupt-driven tasks. However, the predictability of the pSOS$^+$ system makes it well-suited to supporting the RTS polling mechanisms as well.

Each of the operating-system-related sources of uncertainty discussed above is addressed by a feature of the pSOS$^+$ development system. For example, pSOS$^+$ provides fixed-time memory allocation and deallocation primitives that would ensure that the RTS processing involved in building a new TAP plan is predictable. Likewise, pSOS$^+$ supports standard socket communication methods, and non-blocking sockets can be set up as described above, to avoid unpredictable communication delays. The pSOS$^+$ scheduling mechanism is preemptive and rigidly priority-driven, so that the highest-priority task is always running on the processor. Therefore, there would be no danger of a pSOS$^+$-based RTS being swapped out unexpectedly, since the RTS could simply be specified as the highest-priority task.

In sum, pSOS$^+$ is an appropriate environment for an implementation of the RTS which would provide the completely predictable performance required to enforce CIRCA's guarantees. Modifications to the current RTS code would be minimal, primarily surrounding the use of sockets and other system calls such as system clock accesses.

[5] The RTS occupies about 550 kilobytes of memory.

6.4 RTS Performance Metrics

Although the absolute speed of the RTS is unimportant, it is certainly true that we would like the RTS to be fast *relative to its environment.* Therefore, when considering the domains to which the current CIRCA implementation is applicable, it is important to consider the speed of the RTS, and the overhead involved in its processing.

For that reason (and not to show that the RTS is "real-time"), we provide several measures of RTS performance, running on a Sun SPARCstation IPC. As noted above, this Unix machine does not provide truly predictable performance, so we characterize RTS performance by average values, rather than worst-case values. In the worst case, the RTS response time may occasionally be orders of magnitude slower than the average value, because of operating system context switches and interference from other tasks running on the host computer.

To measure the absolute speed and overhead of the RTS implementation (compiled with optimization enabled), we ran several hundred thousand iterations of a trivial TAP consisting of a test that always evaluates to `T` and a no-op action. The average time to execute each iteration of this TAP was approximately 70 microseconds. Over several runs of this test, the results varied on the order of plus or minus 5 microseconds, depending on the other loads running on the machine.

To factor out the execution time of the trivial test and action functions, and thus derive the actual overhead of the TAP selection, execution, and monitoring code, we also timed a simple sequence of the same test and action functions used in the trivial TAP. The test and action code took approximately .5 microseconds to execute. The time used by the trivial TAP is thus negligible, and the overhead of the RTS mechanisms can be considered to be approximately 70 microseconds per TAP invocation. The maximum possible frequency of TAP execution is therefore approximately 14200 TAPs per second.

Within the 70 microseconds of RTS overhead time, at least 56 microseconds (or 80%) is used up by the two **gettimeofday()** system calls used to time the execution of each TAP to decide whether slack time is available for if-time TAPs. Clearly, minimizing the cost of such system calls should be a major goal of real-time operating systems development for applications in which timing information is required. In fact, the $pSOS^+$ system discussed above provides much faster constant-time access to clock services, requiring 15 microseconds to get the current time on a 20MHz 68020.

Similar RTS tests were used to measure the overhead involved in the code that looks for an if-time TAP to execute when slack time is available. This code also contains two **gettimeofday()** calls, to record the amount of time used by the process of finding and executing an eligible if-time TAP. This code, very similar to that which selects a guaranteed TAP, also required about 70 microseconds.

Based on individual timings of the TAP schedule downloading process for schedules of several different sizes, the overhead cost of reading in and parsing the new schedule is about 30 microseconds per character. Thus the time used by the **get-new-schedule** TAP on each invocation can be easily controlled by varying the number of characters it will read in. For most of the experiments in this dissertation, the TAP was allowed to read in up to 500 characters per invocation, limiting its run-time to approximately 15 milliseconds.

These figures for RTS performance should not be misinterpreted to indicate that our example applications have run at a rate of thousands of TAPs per second; rather, these low overhead figures are meant to show that, in our example domains, the domain-specific processing required for TAP tests and actions is the dominant factor, and far outweighs the RTS overhead. For example, in the simulated Puma domain, each primitive that must communicate over a socket with the simulator requires nearly .04 seconds to execute, several orders of magnitude longer than the RTS overhead. In the Xwindows demonstration domain[6], the fastest-response domain we have investigated, the RTS executes on the order of 20 TAPs per second (although, as described in Section 7.1.1, this rate is much slower than necessary, in order to make the user interaction have more obvious effects).

Absolute speed measures of the RTS implementation are useful only as a guideline in choosing domains to which this system may be applied. The RTS we have described can run hundreds or thousands of TAPs per second at best, and thus it is not suited to domains requiring nanosecond response times. However, for many domains in which the system will control physical devices such as robots, the speed of the current RTS implementation is more than adequate, because the slow domain speeds and the inertial effects of mass make control frequencies above 10 to 100 cycles per second unnecessary. Likewise, applications requiring communication with sensors or human users have limits on the required interaction frequencies that should be within the current RTS' capabilities.

[6]Described fully in Section 7.1.1.

CHAPTER 7

EVALUATION: TRADEOFF METHODS

The goal of evaluating CIRCA is not to demonstrate quantitative improvements over the performance of traditional AI, reactive, or real-time systems. We do not wish to show that CIRCA runs faster than previous systems, or uses less memory, or other such implementation-dependent measures. Rather, our goal is to show that CIRCA provides performance capabilities that are fundamentally, qualitatively different from those previously available.

We have already proven that CIRCA combines the ability to run arbitrarily complex, unpredictable AI methods with guaranteed, predictably real-time performance of critical control tasks. One particularly interesting aspect of CIRCA's introspective nature is that it performs its own plan evaluation and verification; that is, if the Scheduler and AIS are able to build a complete TAP schedule, then CIRCA guarantees all of its control-level tasks, and no external performance proofs are necessary. Assuming that the system is given correct descriptions of its primitive capabilities and the environment, CIRCA automatically derives correct, fully-scheduled control plans whose behaviors are well-understood.

Thus, if the system is given sufficient resources to guarantee all of its goals, there is very little to evaluate. Instead, our evaluation of the CIRCA implementation and its collective behavior will focus on the system's abilities when resources are overconstrained. By reasoning explicitly about its own guaranteed behaviors, CIRCA is able to make "conscious" tradeoffs along performance dimensions which have been inaccessible to previous intelligent control systems. This ability is fundamentally different from other systems which base their performance tradeoffs on estimates or experimentation. Because CIRCA is able to explicitly and accurately reason about its own predictable performance, it can not only recognize overconstraining domains, it can also analyze the potential effects of various changes to its goals or plans.

To demonstrate the CIRCA tradeoff mechanisms and show how they provide qualitatively unique performance, this chapter describes the results of several experiments in which CIRCA makes different performance tradeoffs, automatically yielding behaviors tailored to the available resources. Note that these tradeoff methods are not heuristics themselves; they can be implemented by simple procedures making bounded changes to the CIRCA data structures describing the world model, the current control plan, etc. Furthermore, the effects of those changes are well-understood; CIRCA can explicitly reason about the impact of applying its various tradeoff methods on the system's performance. However, choosing which tradeoff method to apply in a particular situation remains a heuristic decision we have not yet addressed.

The experimental data in the following sections was produced using several variations of the basic Puma domain, as detailed in each discussion section. The complete description of the basic Puma domain is listed in Appendix E, including the timing values used for each of the RTS primitives.

7.1 Tactical Tradeoffs by the RTS

As discussed in Chapter 2, CIRCA is designed to implement both tactical and strategic performance tradeoff methods, by combining the strengths of the real-time subsystem and the AI subsystem. Tactical, run-time tradeoffs are the responsibility of the RTS, which dynamically adjusts its use of resources to meet the current state of the world. In this section, we will demonstrate two ways the RTS implements these tradeoffs.

7.1.1 Tradeoffs via If-time TAPs

The if-time TAPs executed by the RTS can allocate resources that only become available at run-time. To demonstrate the if-time TAP capacity for dynamic tradeoffs, we performed tailored experiments in the simple "bouncing box" domain, shown in Figure 7.1. This domain provides an interactive, highly visual, and intuitive illustration of the if-time TAP mechanism. In this graphical simulation domain, the CIRCA control system is responsible for meeting two hard deadlines. First, the system must bounce the left, filled box around the graphics window, moving the box a small amount at least once every .05 seconds. This simple, regular task emulates many types of "heartbeat" real-time tasks that do not vary over time. The second real-time task is more dynamic: whenever the mouse-controlled cursor is moved in the window, the control system must track its motion by drawing a circle around it, at least once every .05 seconds. This task has the visual effect of producing a circle that smoothly tracks the cursor motion. The cursor-tracking task represents a dynamic, unpredictable load on the system which may vary at run-time between requiring no resources (when the cursor is not moving) and requiring some fixed maximum amount of resources (used to draw the circle). To take advantage of the resources that may thus be made available at run-time, the domain also includes an optional task of bouncing the rightmost box, drawn hollow to distinguish it from the box which must be bounced regularly. The complete domain description given to CIRCA is listed in Appendix E.

By implementing the optional box-bouncing operation as an if-time TAP, CIRCA is able to take advantage of the dynamic nature of the environment. Figure 7.2 shows how many iterations of the individual TAPs for each task were executed under varying loads of cursor motion, created by a human user moving the mouse for varying amounts of time.

As shown in the graph, when the cursor was not moving at all, the optional box-bouncing TAP was executed nearly as many times as the mandatory, real-time box-bouncing TAP[1].

[1]There are actually slightly fewer if-time TAP executions because occasional Unix delays decrease the amount of slack time available below the amount needed for the if-time TAP. See Section 6.3.6 for more.

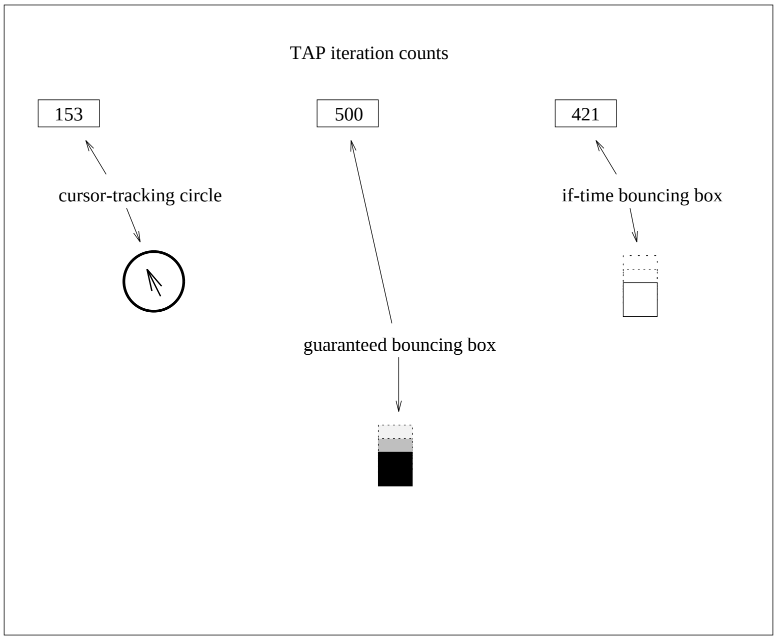


Figure 7.1: The simple "bouncing box" domain.

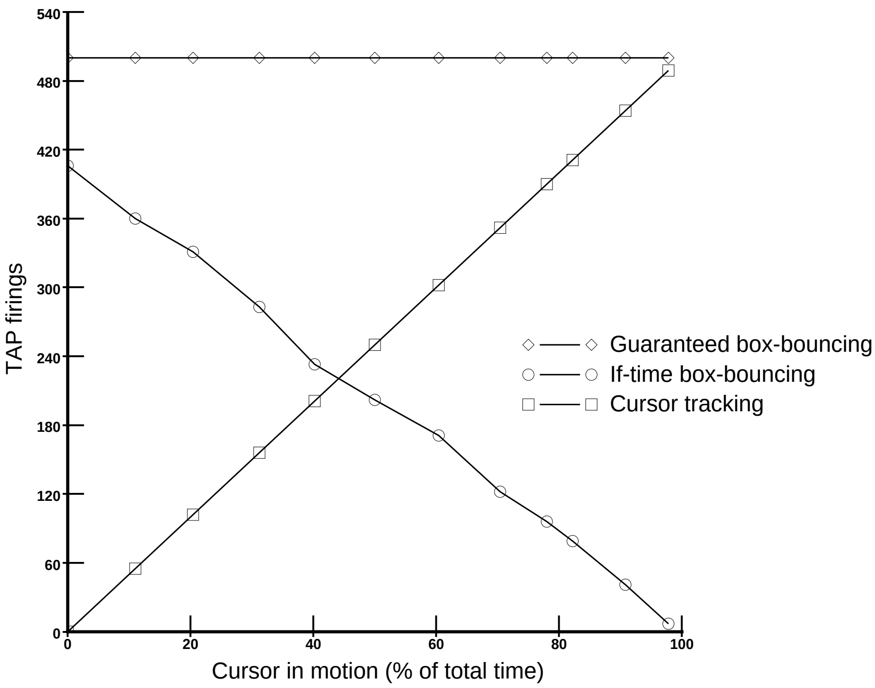


Figure 7.2: Performance results as an if-time TAP makes tactical tradeoffs.

However, as the amount of cursor motion increased, the cursor tracking TAP used up its allocated time more frequently, and the optional box-bouncing TAP was able to use slack time less frequently. At the extreme, when the cursor tracking TAP was executing on every iteration, the optional TAP was completely shut out, and received no execution resources at all. Thus if-time TAPs can implement a form of any-resource algorithm (see Section 2.2.2), using as much resource (here, computation time) as available, but providing no guaranteed performance quality (here, the if-time box stops entirely when the cursor is moving too much).

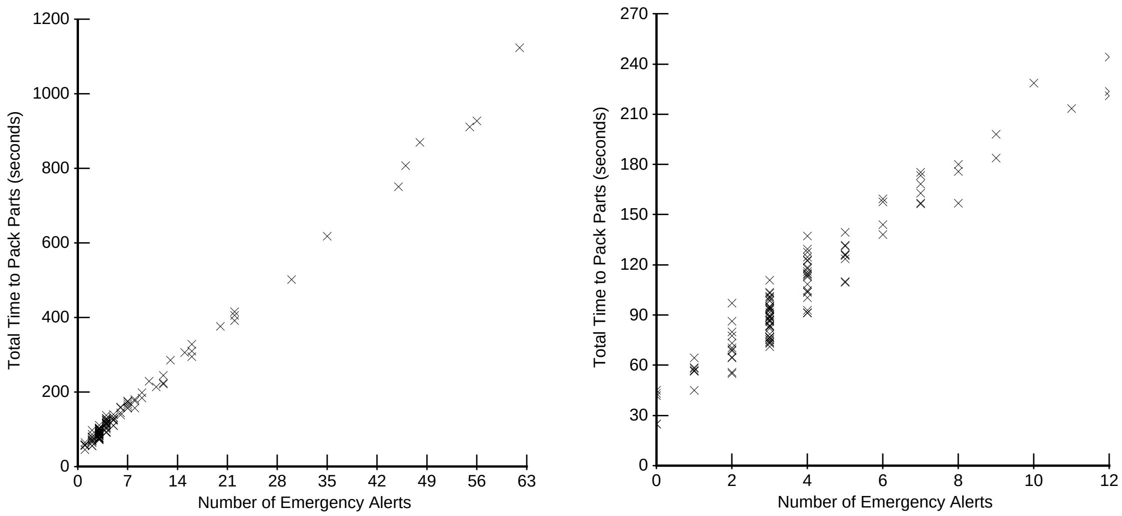
It is interesting to note that the primitives used in this domain are much faster than actually necessary to meet the deadlines assigned. We added artificial delays to the primitives and chose the relatively long deadline timings for two reasons. First, the longer timing values reduce the relative magnitude of the timing variations introduced by Unix (see Section 6.3.6). Second, the slower primitives make the user's interaction (changing the mouse position) have more significant effects on the behavior of the system. If the primitives are used without added delays, the RTS can execute several hundred TAPs each second, so only a small percentage of the cursor-tracking TAPs will ever be executed to actually track the cursor (because Xwindows does not update its readings of the cursor position quickly enough).

7.1.2 Tradeoffs via Planned Behaviors

The RTS can also implement dynamic performance tradeoffs simply by executing the reactive plan sent to it by the AIS. This plan itself may specify how to make tradeoffs in behaviors based on conditions that can only be determined when the plan is being executed. Each step of a reactive TAP plan is conditioned on various tests, and some planned processes that are initiated by TAPs may be interrupted, halted, and resumed, whenever necessary. These interruptions are planned behaviors, implemented as separate TAPs, but their effect is to make the RTS dynamically assign resources (such as the Puma arm) to different tasks depending on the environment.

To demonstrate this capacity in the Puma domain, we disabled the conveyor belt and initialized the simulation with four parts already queued on the table, waiting to be packed in the box. The conveyor was eliminated to avoid the complicating effects of newly-arriving parts. Both the TAPs performing the packing operations and the TAPs that respond to the emergency alert light were put onto the guaranteed TAP schedule. This does not mean that the entire packing sequence was guaranteed to succeed, but rather that the several TAPs required for that sequence were definitely being executed periodically, as opposed to in an if-time manner. This has the effect of isolating the planned behaviors implemented by the TAPs from the complicating dynamic effects of the if-time mechanism. So, in this modified domain, the Puma must try to pack the waiting parts into the box, but that behavior may be interrupted by emergency alerts. To demonstrate the long-term tradeoff behavior of these planned TAPs, we varied the arrival rate of alerts, and measured the effect of that parameter on the total time required to pack all of the waiting parts.

Each time an alert arrived, the Puma would have to make sure its gripper was empty (by putting back down a part, if it had already picked one up from the table), and then move to



(a) Full data set.

(b) Zoomed-in portion.

Figure 7.3: Trading off packing parts for emergency responses.

push the emergency button. Once the button was pushed and the emergency cancelled, it would immediately resume the process of picking up and packing the parts from the table. As shown in Figure 7.3, the RTS behaved as expected, packing the parts more quickly when fewer alerts arrived during the packing task. The zoomed-in graph in Figure 7.3b shows that there was considerable variation in the actual amount of time required for the packing task given any particular number of interrupts. This variation is the result of the differing costs of responding to interrupts that occurred during various phases of the part-packing plan— if the interrupt occurs before the robot has picked up a part, the response time can be faster, and the part-packing will be resumed more quickly. If a part is already being moved towards the box and the interrupt arrives, the robot must move back over the table stacking area, put the part down, and then push the button.

This example clearly illustrates the system's ability to make tradeoffs in run-time performance based on planned reactive behaviors. Because the system is able to interrupt the ongoing execution of the part-packing series of reactions, in order to respond to the more-urgent emergency alert, it trades off the timeliness of achieving the task-level goal of filling boxes for the timeliness of responding to control-level emergencies and preventing failure.

7.2 Strategic Tradeoffs by the AIS: Being Intelligent About Real-Time

Given the truly predictable, real-time performance of the RTS demonstrated in Section 6.3, CIRCA's major innovation is its ability to reason about, design, and adjust that real-time performance based on its analysis of the domain: that is, CIRCA is "intelligent *about* real-time." To illustrate and evaluate this capability, we have designed a number of experiments that show how CIRCA can respond to constraints on its environment and its resources by making explicit, intelligent tradeoffs in the behaviors it implements on the RTS.

CIRCA currently has several ways of recognizing that the domain is overconstrained, and that the system cannot guarantee all of its control-level goals. During the action-planning phase, CIRCA may finish a complete search of the space of possible reaction plans and find that there are no suitable plans that can prevent failure. In that case, the TAP planner will essentially backtrack off the top of its stack, and the AIS interpreter can trap this error and recognize the problem. Or, if the AIS spends too much time trying to build a TAP plan or schedule, it may time-out and be alerted by the timer interrupt described in Section 5.2.1. Finally, the TAP planner may come up with a set of desired TAPs which are then rejected as unschedulable by the Scheduler. This is the most common way of recognizing an overconstrained domain: a suitable TAP plan exists, but it cannot all be guaranteed. At this point, the standard TAP planning method would backtrack to make a different choice and produce a modified TAP plan. Alternatively, the AIS might decide to make a tradeoff instead, somehow easing the scheduling problem to make the current TAP plan more acceptable.

In response to any one of these signals that a particular domain is proving difficult, CIRCA may make one of several tradeoffs. The following sections describe the tradeoff methods that have been implemented on the prototype CIRCA system. The tradeoff methods are generally cast as either alterations to the world model used for planning TAPs, or as changes to the TAPs themselves. For each of the main tradeoff techniques, we first provide details on the type of changes being made to the system, and a general description of the expected effects. We then describe experimental results from an example application of the tradeoff method, and we generalize these experimental results to examine the broader issues relating to the tradeoffs, including what types of information are required, and when each particular tradeoff might be appropriate.

7.2.1 Ignoring a Temporal Transition to Failure

We have identified and implemented several methods by which the AIS can modify its world model to account for resource limitations, so that it builds TAP plans that make various types of performance tradeoffs. These modifications correspond to the various types of transitions (temporal, event, and action) used in the model structure. We begin by describing the planning-time tradeoffs achieved by simply deleting or ignoring one or more temporal transitions that lead to failure in the world model. This corresponds to the planner not even considering that some ongoing process will ever lead to failure. As a result, the TAP that was planned to preempt that temporal transition to failure (TTF) may be affected in several ways. In the following material, we will examine in detail one example of the types of performance tradeoffs that result from simply ignoring a TTF. The experimental results for this first example are particularly lengthy because we describe several aspects of the domain that initially interfered with the desired behavior. We then outline several other possible outcomes, but do not investigate them in depth because they represent minor variations.

One Possible Result of Ignoring a TTF

If the AIS ignores a TTF from a particular state, it is possible that the AIS will still choose the same action for that state, but that the action will no longer be preventing a failure. Because the action is not preempting a TTF, it will be implemented by an if-time TAP. Thus, ignoring a TTF can have the net effect of moving a TAP from the guaranteed list to the if-time list, making the scheduling problem easier. However, performance will suffer because the system no longer guarantees to execute the affected TAP.

Assuming that the relaxation of the TAP scheduling requirements due to ignoring the TTF makes a schedule feasible, this change to the world model effectively voids the guarantees that CIRCA was previously trying to ensure. The system will no longer guarantee to avoid the failure led to by the ignored TTF. However, because in this case an action was planned even without the TTF, the system will still avoid failure whenever the if-time TAP implementing the planned action is able to fire and preempt failure.

In the Puma domain, for example, the AIS might decide to ignore the possibility of a part falling off the conveyor, perhaps because it is highly unlikely that the part will really fall. As a result, when examining a state in which a part is waiting on the conveyor, the AIS will no longer be required to plan a **pickup-part-from-conveyor** action to avoid failure. However, the action will still be planned because it is useful in achieving the system's goals: the robot must pick up the part in order to pack it in the box, which satisfies the goal **(part-in-box T)**.

Schedulability Effects of Ignoring a TTF

Figure 7.4 shows the effect on schedulability for a range of arrival rates for emergency alerts and parts. If the arrival rates match a point below the lower, "normal plan" curve, then the system can build a schedule that will guarantee to both avoid emergency failures and prevent parts from falling off the conveyor. The form of this curve illustrates the tradeoff that the scheduling mechanism can make between tasks; when the emergency rate is relatively high, the system will still build a schedule, as long as the part arrival rate is sufficiently low that the Scheduler can allocate more resources to the tasks that respond to the alert. Conversely, when the emergency rate is lower, the system can deal with a faster rate of arriving parts. If the arrival rates match a point above the lower curve, then the system cannot build a schedule that will guarantee to avoid both emergency failures and dropping parts. However, if the system ignores the **part-falls-off-conveyor** TTF, then it can build guaranteed schedules for all of the instances below the upper line, the maximum rate of emergency alert arrivals that can be handled with the given primitives. The part arrival rate is no longer critical to the scheduling problem, because the **pickup-part-from-conveyor** TAP, with a period determined by the **part-falls-off-conveyor** TTF, is no longer being scheduled and guaranteed.

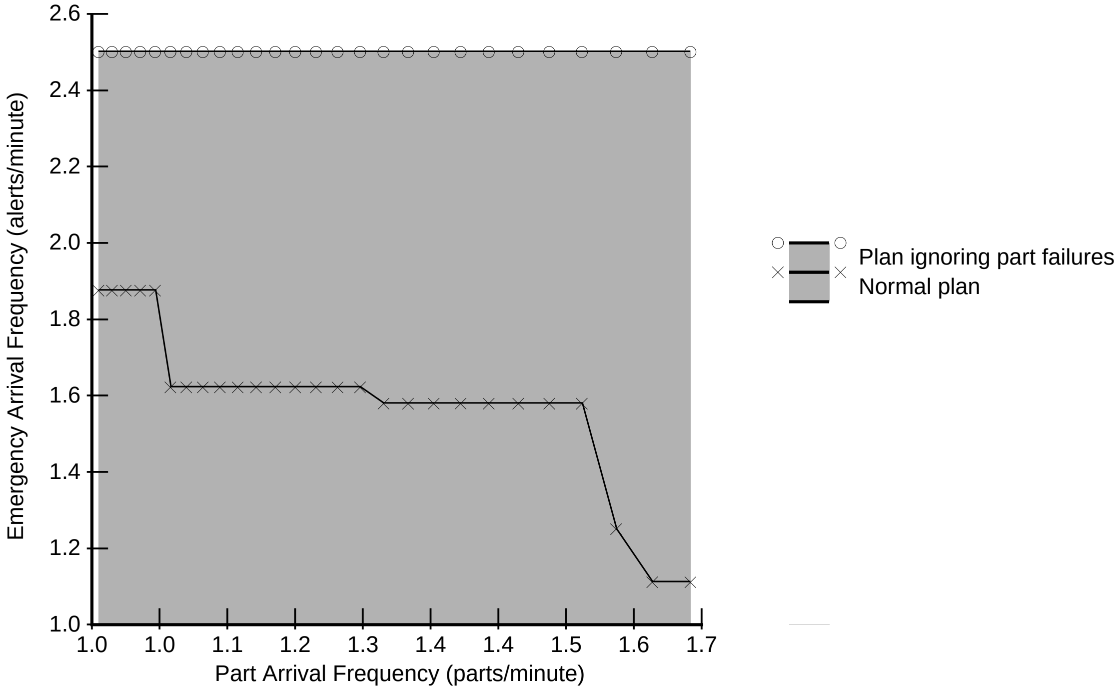


Figure 7.4: The improved schedulability achieved by ignoring a TTF.

Performance Effects of Ignoring a TTF

To illustrate the non-guaranteed nature of the resulting behavior, we implemented this tradeoff method in the Puma domain, increasing the rate of emergency alerts and part arrivals so that the original plan of actions is not schedulable. The AIS then removes the **part-falls-off-conveyor** TTF from the world model, re-plans, and builds a new TAP plan in which the **pickup-part-from-conveyor** action is implemented by an if-time TAP rather than a guaranteed TAP. We expected that, as parts and emergency alerts arrived more frequently, the number of parts falling off the conveyor would increase, as the system had less and less free time to apply to if-time behaviors.

Interestingly, our initial experiments with this method of modifying the world model revealed an aspect of the resulting behavior which we did not anticipate. Figure 7.5 illustrates the behavior displayed by the first few test runs, comparing the number of failures (due to parts falling off the conveyor) with the arrival rate of parts. For these experiments, the delay between emergency-alert arrivals was fixed at 25 seconds, and the failure count was collected after eight parts had arrived and either fallen off of or been removed from the conveyor. As shown in the figure, the number of dropped parts is not strongly correlated with the rate of part arrivals, although at the higher arrival rates there is a tendency to have more dropped parts. Still, even at the lowest arrival rate shown, there were instances where many of the arriving parts fell off the conveyor. The cause of this behavior is not an aberration in the if-time TAPs or some other fault; rather, it is a result of a choice that was made during the planning phase.

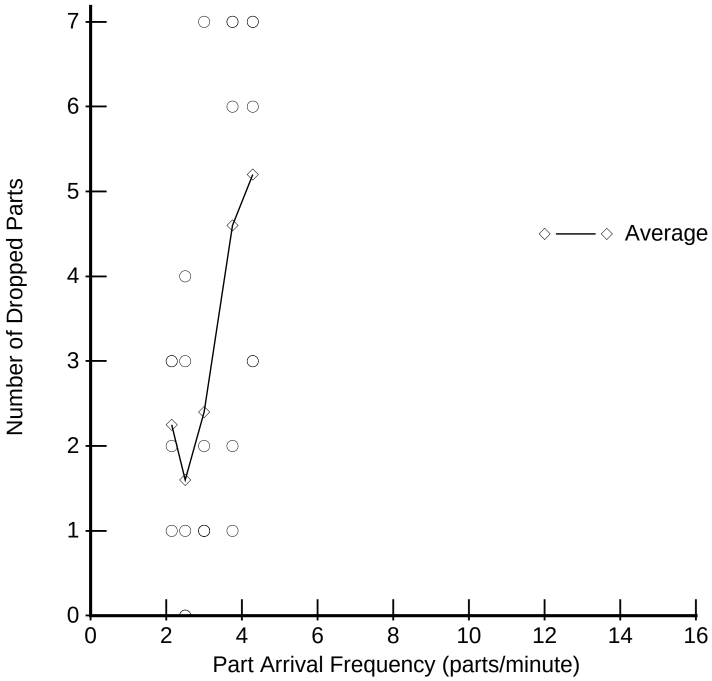


Figure 7.5: The non-guaranteed, if-time behavior resulting from ignoring the **part-falls-off-conveyor** TTF. Note that circles represent more than one data point. The scale was chosen to match later graphs, for ease of comparison.

When the planner was no longer required to guarantee that parts would be removed from the conveyor before a deadline, the goal of having no parts on the conveyor [**(part-on-conveyor NIL)**] became equally ranked with another goal, that of having no parts left on the table [**(part-on-table NIL)**]. As a result, when the planner chose an action for a non-emergency state in which parts are available on both the conveyor and table, neither part was preferred for packing into the box. Using arbitrary ordering information to resolve this tie, the planner initially chose to pick up and pack the part from the table. With the emergency alert arrival rate set near the maximum possible response rate, the system was frequently interrupted by an emergency while trying to pack a part from the table into the box. It would then replace the part on the table, push the emergency button, and return to the part on the table. This looping behavior, once started, left the system unable to respond to any subsequently-arriving parts on the conveyor belt. Thus, no matter what the frequency of part arrivals, once the system entered this pathological cycle it was unable to respond to later arrivals. Note that this behavior is not erroneous: the planner was explicitly told that it no longer needed to guarantee to avoid dropping parts, by the removal of the TTF. Thus, without any other preference knowledge, parts off the table or off the conveyor are equivalent.

To resolve the tie between picking up parts on the table and on the conveyor in a more rigorous fashion, we simply employed the repeat-goals mechanism in the planner (see Section 5.3.1) to add priority to the **(part-on-conveyor NIL)** goal. This makes picking up parts from the conveyor yield the extra benefit of achieving a repeat goal, so the planner selects that action over picking up a part from the table, when both are applicable.

Even with this change in the TAP plan, the system still displays unusual behavior, as shown by the data in Figure 7.6. Observations of the modified TAP plan actually running revealed another source of fluctuations in the number of parts dropped. Because the emergency alert arrival rate was fixed at a constant value, it was possible for the phases of the periodic part arrivals and emergency alerts to match in both beneficial and detrimental ways. At some times, the arrivals rates would be “in sync,” so that the robot would finish packing a part just as an emergency alert arrived. Because the RTS did not need to perform any cleanup actions (such as putting a part down on the table), the emergency received a rapid response, and thus when the next part arrived there was still some delay before the next emergency could occur. The sharp dip in Figure 7.6 at a part arrival frequency of 5 parts per minute is caused by precisely this effect: the arrival rate of parts and emergencies were nearly on harmonic frequencies (12 seconds between parts, about 25 seconds between alerts). The relatively short duration of this test (only 8 parts arriving) allowed the system to remain in the beneficial portion of the synchronization for long enough to pack almost all of the parts, despite the lower quality of performance achieved at both immediately higher and lower part arrival rates.

In other situations, the arrivals could be synchronized in a detrimental way, so that the emergency alert would arrive immediately after the robot had just picked up a part. As above, this leads to an extended sequence of actions to respond to the emergency, thus squandering the delay before the next emergency, and making it more likely that the next part picked up will also not be packed.

As with the previously-described behavior, these synchronization effects are not errors—in fact, the detrimental synchronization described above is precisely the sort of condition used to derive the worst-case performance of TAPs (see Section 4.3). Thus the guarantees made by the planner still hold in these situations. Since the planner has given no guarantee that it will avoid dropping parts off the conveyor, it is simply doing its best to avoid these problems. These results show that it might be very useful for a system to have a way to recognize and avoid such pathological synchronizations; a small added delay or altered decision could dramatically alter performance in some cases. For example, if the system recognized that it had arrived at the emergency alert button only after a relatively long delay, and thus another alert might be imminent, it could hesitate near the alert button, ready to respond quickly and then switch to packing parts during the subsequent alert-free period.

To nullify the synchronization effects and show more clearly that the revised plan performs better in less-heavily-loaded situations (when the if-time TAPs can more frequently pack parts), we modified the simulation environment so that emergency alerts arrived with random delays uniformly distributed in the range of 25 to 30 seconds. With this change to the environment, the system displays behavior more in line with our intuitions, as show in Figure 7.7.

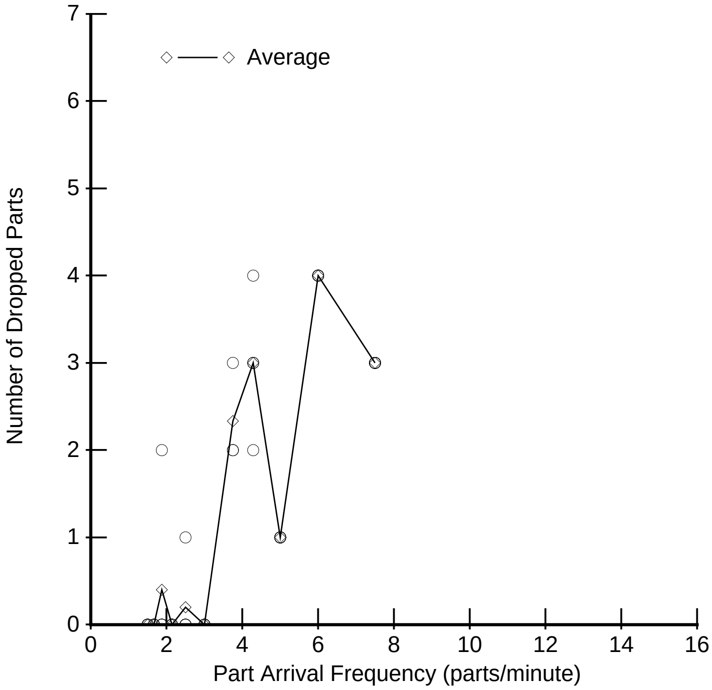


Figure 7.6: The revised behavior resulting from ignoring the **part-falls-off-conveyor** TTF and prioritizing parts arriving on the conveyor.

Generalizing Experimental Results

In any-dimension algorithm terms, this modification of the world model has traded off many aspects of the system's performance in exchange for guaranteeing a subset of reactions. The timeliness and completeness of the reactive system's behaviors (i.e., the guarantee that it will respond correctly, and in time, to all situations) has been lost, because there are some states for which its reactions are not guaranteed. However, it is notable that the system can still make guarantees about some of the world states: it can, for example, guarantee that it will avoid failures resulting from the emergency alert, because the actions preempting the **emergency-failure** TTF are still guaranteed.

Thus this technique of ignoring a TTF in order to reduce the resource requirements of a particular environment is useful in situations where the resulting loss of guarantees is tolerable. There are two reasons that CIRCA might decide this is the appropriate method:

1. The TTF to be ignored represents a process that rarely actually operates at its worst-case rate, so in all likelihood the RTS will prevent failures even with an unguaranteed, if-time TAP.
2. The failure mode which the TTF leads to may be reconsidered and treated as non-catastrophic. In the Puma example, this would correspond to modifying the **part-falls-off-conveyor** transition to lead instead to a non-failure state, possibly distin-

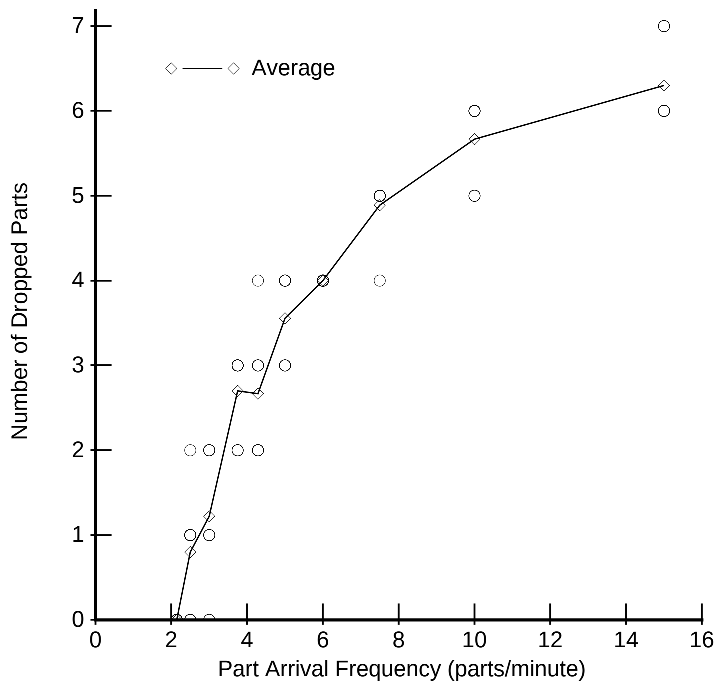


Figure 7.7: The final behavior after ignoring the **part-falls-off-conveyor** TTF, prioritizing parts arriving on the conveyor, and randomizing emergency alert arrivals.

guished by an additional feature such as **(part-on-floor T)**.

Alternative Results of Ignoring a TTF

Several other outcomes are possible when the AIS chooses to ignore a TTF. For example, it is possible that, by ignoring one TTF from a state, a different temporal transition becomes dominant and still causes the planned action for that state to meet a deadline. In general this will mean that the $min\Delta$ for the planned action will be longer, but the TAP will still need to be scheduled. The resulting tradeoffs are similar to those above, in that the system can no longer guarantee to avoid all types of failures. In this case, however, no TAPs are moved out of the guaranteed list: instead, the `MAX-PERIOD` of one of the TAPs will be increased, thus decreasing the desired utilization, and making the scheduling problem easier.

In the Puma domain, for example, the periods of many of the TAPs are constrained by both the part arrival rate and the emergency arrival rate. Consider, for example, a state in which the emergency light is on, the robot is holding a part, and a part has arrived on the conveyor belt. An abstracted version of one such state is illustrated in Figure 7.8. Both the **emergency-failure** TTF and the **part-falls-off-conveyor** TTF are applicable

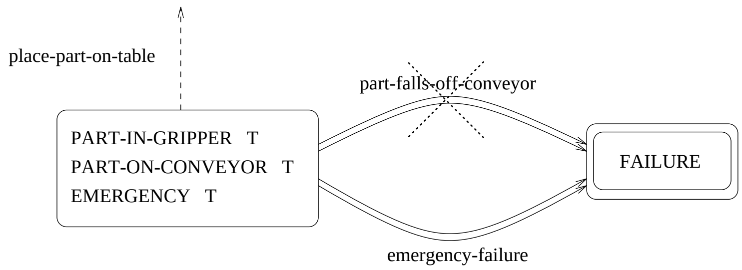


Figure 7.8: Removing a TTF may just alter a TAP's period.

to this state. In response to these threats of failure, the planner will build a TAP to place the part from the robot's gripper onto the table as quickly as possible. If the $min\Delta$ of the **emergency-failure** TTF is fairly long, then it is possible that the tightest constraint on the planned TAP will be derived from the $min\Delta$ of the **part-falls-off-conveyor** TTF. Now, if the **part-falls-off-conveyor** TTF is ignored because the AIS has decided to make a performance tradeoff, the same action will still be planned for this state— it is still necessary to avoid failure due to the **emergency-failure** TTF. However, now the TAP's `MAX-PERIOD` will be increased, because it is derived from the longer $min\Delta$ of the **emergency-failure** TTF.

Because this result of ignoring a TTF keeps all the same TAPs on the guaranteed schedule, it is preferable to the previous example in which a TAP was moved to the if-time list. Leaving the TAP guaranteed means that it is still assured of being run at some frequency, albeit lower than that required by the ignored TTF. In the previous case, the affected TAP may never be run at all, if slack time is not available.

Another slightly different result is possible if the action that was planned to preempt the ignored TTF is also used to preempt some other TTF out of some other state. As above, the $min\Delta$ for the action may be relaxed if the ignored TTF was the dominant, shortest deadline for which the action was planned. In addition, however, the number of domain states for which the action transition is planned will be reduced, and the complexity of the TAP implementation may be either increased or decreased.

The complexity of the TAP test expression may increase (despite intuition) because of the test minimization operation (see Section 5.3.2). The decision tree formed by ID3 may be more complex if a smaller set of positive cases (states where an action is applicable) is given to the algorithm. With a larger set of positive examples, it is possible that the system would be able to find a more-general, shorter set of tests which would suffice. With a smaller set of positives, it may be necessary to include more feature tests to increase the selectiveness of the TAP. As a result, the exact effects of ignoring a TTF in this situation are hard to predict. While the schedulability of the system will generally be improved, because the period of the TAP is increased, the potential for increasing TAP tests may offset that improvement. More costly tests may make the affected TAP more difficult to schedule.

In the extreme case, ignoring a TTF may cause the planner to completely eliminate one or more planned actions, thus removing TAPs from the list to be scheduled. If an action was only planned originally to preempt failure (or as a "precursor" to that preemption), and was not instrumental in achieving any other system goals, then the action may be removed entirely. For example, if CIRCA ignores the **emergency-failure** transition in the Puma domain, it will completely alter the world model and avoid planning the **push-emergency-button** action. Depending on the frequency of part arrivals, it may also eliminate the need to put parts on the table temporarily, and thus ignoring this one TTF could also remove the **stop-moving** and **place-part-on-table** actions from the plan. These latter actions are precursors that were included in the plan to establish the preconditions of the action that was planned to preempt the TTF, and thus they are also unnecessary.

While moving a TAP to the if-time list means that, in non-worst-case situations it may be still executed quickly enough, deleting a TAP altogether provides no such potential. Since if-time TAPs do not use any resources when the RTS is pressed for time, avoiding building if-time TAPs does not save any significant RTS execution-time resources.

The only additional benefit of not building a TAP is that it saves AIS planning time. Because of that effect, this tradeoff method can be usefully applied if the AIS has timed out during the planning process because the domain is simply too complex.

Summary of Ignoring a TTF

We have seen how the simple world model change of ignoring a TTF can have dramatic, varied effects on the overall behavior of a CIRCA system. In particular, this modification of the model can result in the following direct effects:

- Moving a guaranteed TAP to the if-time list.
- Increasing the `MAX-PERIOD` of a guaranteed TAP.
- Increasing or decreasing the complexity of a guaranteed TAP's test expression.
- Eliminating a guaranteed TAP entirely.

An important feature of this tradeoff method, and of the CIRCA approach in general, is that the system can introspectively examine the predicted effects of a particular tradeoff. In other words, CIRCA might evaluate the worth of various tradeoff methods by examining the expected results in the world model. If the AIS considers ignoring a TTF, it can immediately recognize that the failure resulting from that TTF will be possible with the modified TAP plan. In addition, the AIS can examine the new world model and TAP plan to recognize more detailed aspects of the tradeoff. For example, if the new plan still includes all the same guaranteed TAPs as the original plan, then the AIS can conclude that the reaction previously planned to preempt the TTF is still being enforced, but at a lower rate. If the AIS knows that the worst-case rate of the ignored TTF is rarely achieved, this tradeoff option may be very attractive, because it has exchanged a decrease in one TAP's response rate for the ability to schedule and guarantee the entire TAP set.

7.2.2 Ignoring an Arbitrary Temporal Transition

The intuitive motivation behind ignoring TTFs is that it prevents the system from considering some source of failure, and thus prevents the system from committing resources to avoiding that failure. It is also possible to have the planner ignore other temporal transitions that do not lead directly to failure, but may still provide opportunities to lower the required resource usage.

Ignoring non-failure temporal transitions can have most of the effects listed above, because it may make some arbitrarily large part of the world model state space unreachable. If, for example, the temporal transition being ignored is the only connection between the initial states and all of the states for which a particular action is planned, then those states will become unreachable and the action will not be planned.

Ignoring non-failure temporal transitions can have one additional, even more dramatic effect on the planner: it may make one or more goals unachievable. If a critical temporal transition is removed from the world model, there may be no path from the initial states to a goal state. In the Puma domain, if we ignore the **arrive-over-box** temporal transition, which represents the duration of the process of moving over the box, then there is no way for the robot to actually pack a part in the box (because it can never achieve **(robot-position over-box)**). Thus, ignoring temporal transitions related to the controlled agent's behavior is probably not a very useful approach, as it corresponds to reducing the capabilities of the agent, rather than reducing the environmental constraints or requirements.

Since the original motivation for making a performance tradeoff often comes from an inability to schedule guaranteed TAPs, which are generally planned to preempt TTFs, it is frequently more useful to ignore TTFs than other, non-failure temporal transitions. Furthermore, ignoring TTFs is safer in the sense that it cannot make any goals unreachable, because TTFs are never in the path to success.

7.2.3 Ignoring an Event Transition

Just as the AIS may decide to alter its treatment of temporal transitions, it may also choose to change how it considers event transitions. Ignoring an event transition may have many of the effects described above for temporal transitions: it may cut off parts of the world model state space, possibly making some goals unreachable. Ignoring an event transition can thus reduce the planning time and decrease the number of TAPs planned, allowing the system to make guarantees for some subset of desired behaviors which were not previously schedulable.

As with the above tradeoffs, the AIS would only be motivated to ignore an event if it finds that its initial attempts at building a plan are unsuccessful, either because the planning is taking too much time, or because the resulting TAPs are not schedulable. Choosing which event to ignore will generally be a highly domain-dependent decision, possibly based on the system's evaluation of the probability of that event occurring, the benefits of ignoring the event, and the costs of having the event occur when the system has not planned for it.

For example, in the Puma domain, ignoring the **emergency-alert** event transition provides a large reduction in the planning time, because many states are eliminated from the model— in fact, the state space for our running example is reduced from 330 enumerated states and 158 reachable states to 106 enumerated and a mere 58 reachable states. Furthermore, a large number of contingency reactions are eliminated from the plan, and thus the complexity of the TAPs is reduced, and the scheduling problem is eased. Because the emergency alert is no longer of concern, the system is able to react to parts on the conveyor belt even more quickly than if the predicted alert rate is very slow (as in the extreme right edge of Figure 7.4). While the example of Figure 7.4 could handle parts arriving at most every 33 seconds, the plan built by ignoring the **emergency-alert** transition can handle parts arriving every 27 seconds, an 18% improvement in capacity. Of course, the tradeoff is that the system is no longer monitoring the emergency light, and it will not react to an alert. If the AIS thinks that an alert is unlikely, or finds that the cost of failing to respond to an alert is sufficiently low, it may judge that the reduced planning time and improved part-packing reaction time are worth the risk involved in ignoring alerts.

More generally, we can see that ignoring an event transition can have the desirable effects of reducing planning time and simplifying the scheduling problem. The disadvantage, of course, is that this tradeoff method removes planned contingency actions entirely, as opposed to just moving the relevant TAPs to the if-time list (as ignoring a TTF can do). Because event transitions represent instantaneous events in the world, as opposed to the ongoing processes represented by temporal transitions, it seems plausible that the AIS could have knowledge of event probabilities that would be helpful in guiding the use of this tradeoff method. Ignoring highly improbable event transitions would obviously be a good approach, in order to ensure that the system is least likely to encounter world situations for which it is not prepared. Decision-theoretic methods involving expected utility could also be used to account for both the probability of an ignored event and the cost of failing to take the originally-planned action.

7.2.4 Modifying Temporal Transitions

In addition to these drastic methods involving ignoring various transitions, the AIS can make more subtle, deliberate changes to the duration of temporal transitions to effect performance tradeoffs. The basic approach is to extend a temporal transition to give the system more time in which to react and avoid undesirable consequences. For example, the AIS might decide to extend the TTF representing the delay until a part falls off the conveyor, instead of ignoring that TTF altogether. The duration of the TTF, or the minimum time until failure, can be extended in this example by simply slowing down the conveyor, giving the Puma more time to pick up an arriving part.

While the complex physical modeling required to determine the exact relationships between the Puma speed and the conveyor rate is beyond the current AIS' abilities, it is able to make relatively simple decisions about which TTF to modify, and how to achieve that modification. The AIS may learn from the Scheduler, for example, that the TAP which failed to meet its deadline during scheduling was built to implement the **pickup-part-from-conveyor** action. The AIS can easily find that this action was planned to preempt failure resulting from the **part-falls-off-conveyor** TTF. Then, the AIS could find that the minΔ of this TTF may be extended by having the RTS invoke the action **slow-down-conveyor**, which also causes the AIS to modify its model of the relevant transitions. This has the effect of moving the environmentally-driven resource demands to the left in the graph of Figure 7.4, so that the given emergency arrival rate can be handled with the decreased part arrival rate.

This type of tradeoff mechanism is useful when the system can alter the environmental behavior, as in this example, and also when its own behaviors involve temporal transitions whose minΔ values may be altered. For example, an alternative way of dealing with a part arrival rate beyond the system's initial capacity is to increase the speed with which the Puma moves between locations, thus decreasing the minΔ of the temporal transition **arrive-over-box**, which represents the worst-case time the robot needs to move over the box.

On the positive side, this approach to making tradeoffs gives CIRCA the ability to fine-tune its behaviors and the environment to work together well. The very notion that the system can modify the behavior of the environment to make it more convenient for its own goals is relatively uncommon in planning systems, and has recently drawn attention in work by Agre [2] and Hammond [25]. The way CIRCA uses this technique is perhaps unique in that it is motivated by a strong understanding of exactly what the agent is and is not capable of achieving, and thus why the environmental modifications are required in the first place.

On the negative side, this approach requires extensive domain knowledge even beyond the static behavioral information CIRCA already needs. The system must be able to derive a causal mapping between temporal transitions and the parameters that affect their duration, and also decide what actions the agent can take to modify those parameters, in the appropriate way. Qualitative physics [11, 16] might prove to be an excellent way to derive this information.

7.2.5 Modifying TAP Implementations (Method Selection)

In addition to making changes to the world model in response to resource restrictions, the AIS can also make changes directly to the implemented form of the actions planned in the world model. In particular, the AIS can make two major types of changes to the TAPs built to implement action transitions. The first and most powerful modification is to simply alter the specific primitives used to perform the various tests and action required by a TAP. The sensor planning phase of the TAP-generation process implements this functionality, as described in Section 5.3.3. The AIS may have several different methods for performing an action (or a test), and it can choose amongst them according to the resources available. This tradeoff method is equivalent to the "configuration selection" [35], "version selection" [47], and "design-to-time" [19] approaches.

For example, suppose that the Puma control system provides the RTS with two different types of part-placement operations, a slow, high-accuracy, "fine-motion" operation and a faster, lower-accuracy, "coarse-motion" operation. This means that the system has two possible primitive operators for the **place-part-in-box** action transition. Using the fine-motion operator allows the system to place the parts very close together, thus yielding densely-packed boxes. But the fine-motion operator needs four seconds to finish the placement operation. Using the coarse-motion operator requires the system to leave more space between the parts, since the placement is less-certain. As a result, the system will produce less-densely packed boxes, but it can produce them more quickly, because the coarse-motion operator only needs 2.5 seconds. Thus, in this example, method selection allows the system to trade off the quality of its results (the packing density) for the timeliness of its long-term and short-term behaviors (the speed of packing whole boxes and individual parts). Given the faster coarse-motion operator, the system may be able to guarantee to respond in time to a higher frequency of emergency alerts than with the slower operator.

Experimental Results of Method Selection

To provide a more quantitative demonstration of this tradeoff, we ran experiments using the coarse/fine operators described above. The fine-motion operator was defined to require no space at all surrounding parts being placed in the box: essentially, it could achieve 100% packing density with a fortuitous series of part arrivals[2]. The coarse-motion operator, on the other hand, required one inch of clearance on all sides of the parts in order to place them in the box. Naturally, the achievable packing density is lower with this operator, since parts necessarily occupy spaces larger than their actual size.

Figure 7.9 shows the improvement in response-time achieved by using the coarse-motion operator, displayed here by the increased rate of emergency alerts and part arrivals that can be handled. The upper curve shows the response tradeoffs that can be made using the faster coarse-motion packing operator, while the lower curve shows the performance for the fine-motion operator used in the rest of the plans discussed in this chapter (and previously graphed in Figure 7.4). The coarse-motion operator reduces the time allocated to the **place-part-in-box** TAP, and therefore the system can respond in time to more frequent part arrivals, emergency alerts, or both.

However, Figure 7.10 shows the corresponding decrease in performance quality that resulted from the coarse-motion operator, when applied to 100 trials using randomly ordered arrivals of four different part shapes. On average, the density of the packed box was reduced from 70% using the fine-motion operator to 59% with the coarse-motion operator. In these experiments, simulations of the box-packing algorithm were continued until the first arrival of a part that did not fit in the box. The fine-motion version was able to pack an average of 45 parts in the box, while the coarse-motion version packed an average of only 26 parts. Thus we can see that the improved schedulability and response time illustrated in Figure 7.9 are only achieved at the cost of stiff performance degradation.

Generalizing Method Selection

To use the method selection approach, the system obviously must have alternative methods for implementing feature tests and actions on the RTS. In addition, to make intelligent decisions about method selection, the system would require performance information describing the output quality and resource requirements of each method. This information could be relatively simple, or could be as complex as a full performance profile (see Chapter 2). In any case, because method selection retains the consideration of all world model states and does not remove any TAPs from the schedule, it is one of the more subtle tradeoff techniques, capable of altering the resource needs of the system without drastic

[2]None of the packing strategies deliberately reorder the parts by placing them on the table and packing them later. Parts were only put on the table if their shape was unknown, or if the packing operation was aborted to deal with an emergency. In this experiment, no unknown parts were included.

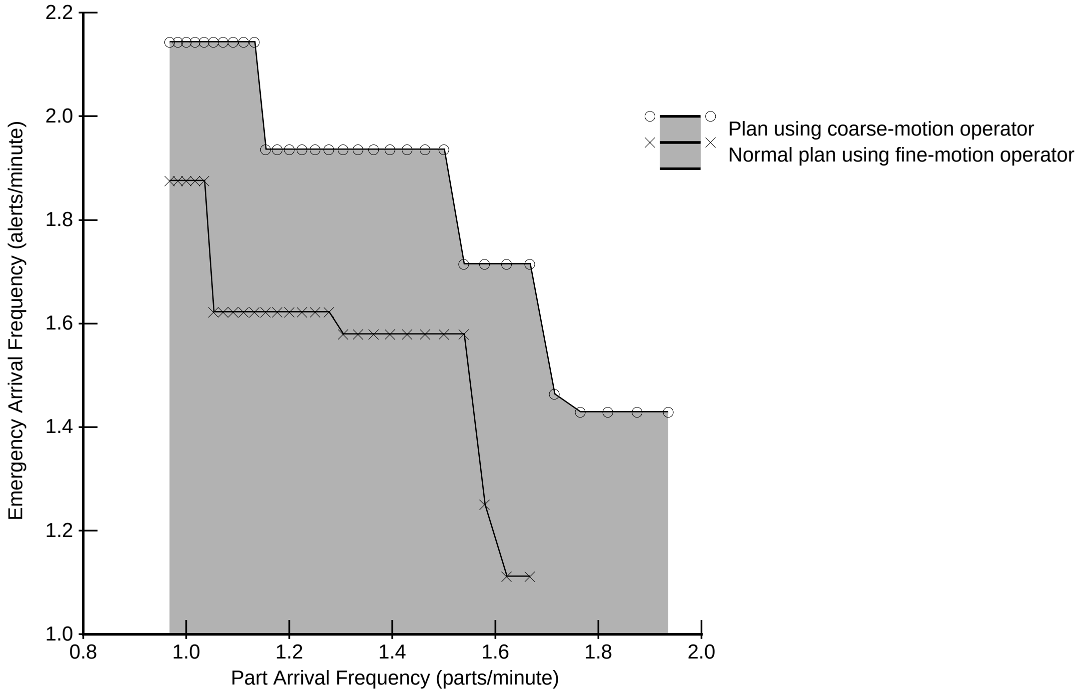
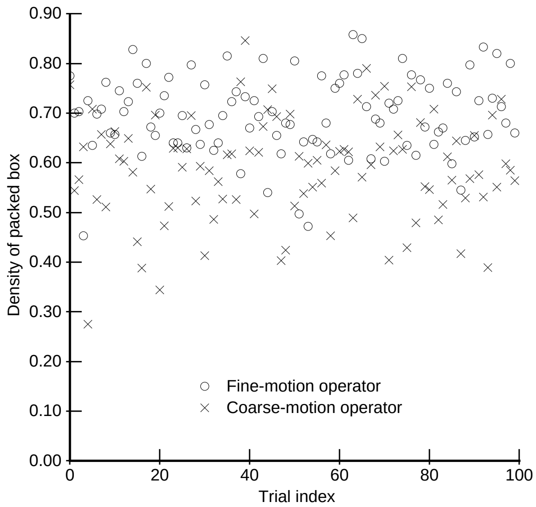
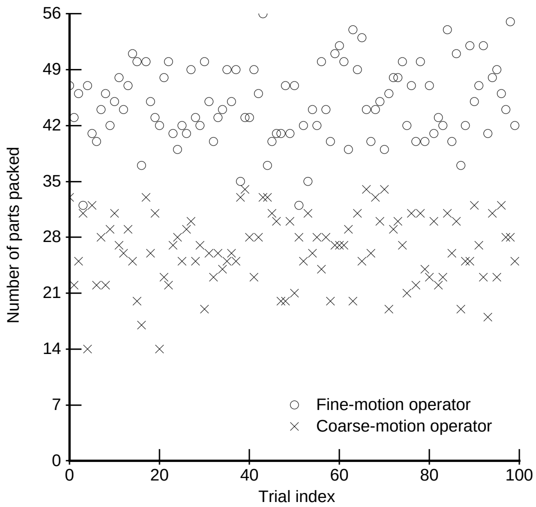


Figure 7.9: Schedulability variations using different TAP implementations.



(a) Density of packed box.



(b) Number of parts packed.

Figure 7.10: Performance variations using different TAP implementations.

effects on its performance guarantees. Depending on the assortment of different methods available, the method-selection approach can alter almost any quality measure of the reactive system's performance, including precision, accuracy, number of parts packed, etc.

7.2.6 Removing TAP Tests

The AIS can also make an unusual type of tradeoff by removing one or more feature tests from the precondition expression of a TAP. Removing an expensive test can make it easier to schedule the TAP, because the resources required for the precondition will be reduced. However, this modification reduces the state discrimination abilities of the TAP, giving the overall plan decreased *confidence*. The AIS is no longer sure that only planned, appropriate actions will be taken by the RTS.

This sort of modification is obviously fairly risky, since drastically inappropriate actions could result. In the Puma domain, for example, the **push-emergency-button** TAP is originally built with the test expression **(and (part-in-gripper nil) (emergency T))**. If the AIS removes the **(part-in-gripper nil)** portion, the RTS may fire this TAP and push the emergency button any time the emergency light is on. If the Puma happens to be grasping a part when the TAP fires, the RTS would obliviously jam the part and gripper into the emergency button, possibly resulting in damage to any or all of these devices.

However, this approach may be useful if the AIS can determine that some expensive test is not used to discriminate between common states, and is only present in the test expression for some rare exception. In that case, the AIS may decide that the benefits of removing the test outweigh the slight risk of taking an inappropriate action.

Removing a test may make a TAP apply to either a larger or smaller set of states, depending on how the particular feature test was invoked within the overall test structure. If the removed test is used in a disjunction, the resulting test expression will apply to a smaller set of states, while if the removed test was used in a conjunction, the new test expression will succeed for a larger set of states.

For example, consider the test expression of the **stop-moving** TAP planned for the Puma domain, shown in Figure 7.11. The original TAP will apply only to states where the robot is moving and either there is a part waiting on the conveyor, or the emergency light is activated. Its purpose is to interrupt the process of carrying a part to the box, so that the robot can quickly place the current part on the table and respond to either the emergency alert or the newly arrived part. If the AIS removes the **(robot-position changing)** test, the TAP is now applicable to a larger set of states; any time a part arrives or the alert light goes on, the TAP may fire, regardless of whether the robot is actually moving or not. Alternatively, suppose the AIS removes all the tests of the **part-on-conveyor** feature. In that case, the reduced test is **(and (robot-position changing) (emergency T))**, meaning that the TAP now only applies when the emergency light is on. Thus the set of applicable states is reduced, since parts waiting on the conveyor will no longer trigger the TAP.

The first case, where the range of applicability is increased, shows how removing a TAP test can lead to unplanned but acceptable actions, rather than just disastrously inappropriate actions. In this case, the system may follow an action loop, taking an action that

```
TAP stop-moving
    :TEST  (and (robot-position changing)
                (or (part-on-conveyor T)
                    (and (part-on-conveyor nil)
                         (emergency T))))
    :ACTION  (stop_moving)
    :MAX-PERIOD 8.84
    :TEST-TIME .18
    :ACTION-TIME .02
```

Figure 7.11: A **stop-moving** TAP for the Puma domain.

has essentially no effect on the world (see Section 4.11). Any time **part-on-conveyor** is true, the system may execute the **stop-moving** action, even if the robot is already halted. In terms of the AIS world model, this would correspond to an action with identical domain and range states, which therefore has no use. The only cost of such an action loop is the overhead required to execute the useless action. Note that there is no danger of the system getting caught in an infinite loop, since the RTS will continue cycling over the schedule, testing and executing the other TAPs as well.

The second case, removing tests of the **part-on-conveyor** feature, leads to an interesting form of behavior because the system has reduced the number of states in which it will halt to interrupt the action of moving over the box. This tradeoff has the effect of prioritizing the completion of the packing operations for a part once it has been picked up. To demonstrate the performance effects of this change, we tested the resulting plan.

Experimental Results of Removing a TAP Test

Extensive simulations show, as illustrated in Figure 7.12, that in some situations the modified TAP has the effect of increasing the number of parts that are successfully packed into the box. To understand this effect, consider the original plan: when the robot was carrying a part over to the box, if another part arrived the robot would immediately halt and place the current part on the table, so that it can pick up the part from the conveyor. The revised TAP disables that behavior, so that any time the robot is carrying a part over to the box, the only event that can interrupt the process is an emergency alert. When the part arrival rate is synchronized so that parts frequently arrive in the middle of the process of carrying the previous part to the box, the original plan will be forced to place more parts on the table than the plan with the modified TAP. Thus Figure 7.12 shows that, at 7.5 parts per minute, the modified plan packs significantly more parts into the box than the original plan (on average). At both lower and higher arrival rates, the two plans perform very similarly, because the world states affected by the change to the **stop-moving** TAP rarely occur. At lower part-arrival rates, the robot is able to finish packing most parts before the next ones arrive. At higher rates, the parts arrive so quickly that the robot never even begins moving them over the box, it just tries to pick them up and put them on the table as quickly as possible.

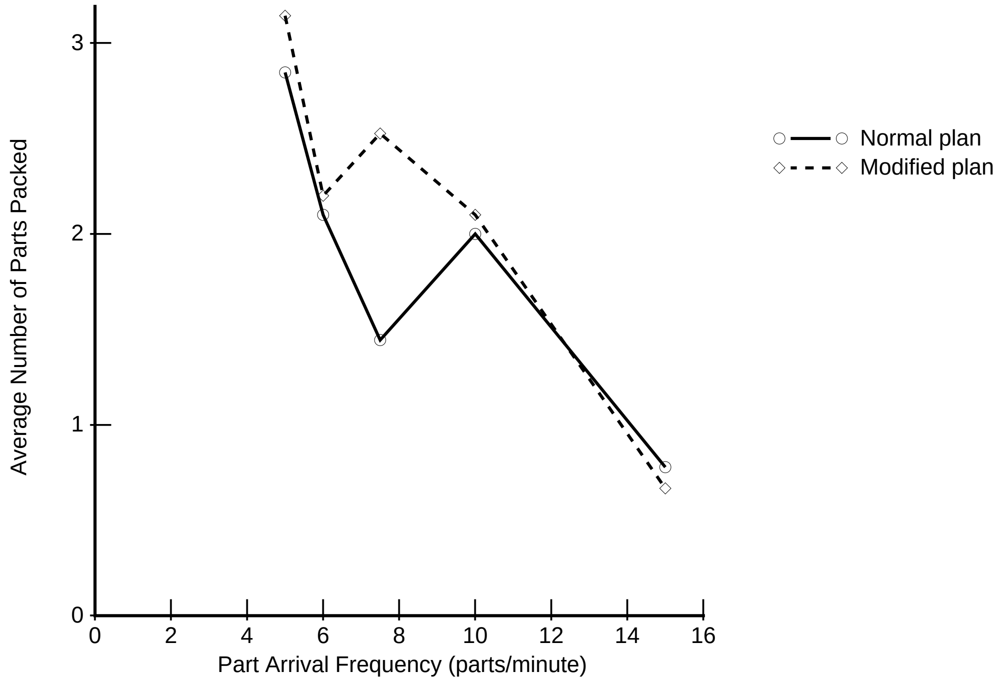


Figure 7.12: Performance effects of removing the **part-on-conveyor** tests from the **stop-moving** TAP.

This is a somewhat anomalous result, showing somewhat improved performance with a partially-mutilated TAP plan. In general, removing a test from a TAP will result in non-optimal behavior caused by inappropriate actions or the failure to take a desired action.

This tradeoff method is clearly most useful when there are some world features that are quite costly to test, so that eliminating those tests is worth the resulting increase in behavioral uncertainty. In the Puma domain, testing is relatively fast compared to the duration of the robot actions, so the schedulability of a TAP plan is hardly affected by eliminating a small number of feature tests. However, if vision processing or other costly sensing methods were in use, this tradeoff technique might prove very effective at decreasing a TAP plan's resource requirements without excessively damaging its performance.

As with the earlier tradeoff methods that ignore transitions, one of the most powerful aspects of removing a test from a TAP is that the effects of this change can be considered in the context of the world model. The AIS could scan over the reachable model states and check when the modified TAP would be applicable, comparing those occasions with the originally-planned states. This would allow the AIS to determine whether the TAP would be applied to inappropriate situations, and whether those applications could lead to serious problems. Similarly, the AIS could recognize when the revised TAP would fail to implement a critical transition because it would no longer apply to the appropriate state. Using this capacity to project and examine the results of a proposed TAP modification, the AIS could make sound decisions about what types of performance tradeoffs would provide a good match of maximum increased schedulability with minimal decreased schedule confidence.

7.3 Summary

In this chapter we have investigated a variety of ways in which CIRCA can make performance tradeoffs in the face of resource limitations. This capability is a fundamental requirement for intelligent real-time systems, since the very nature of real-time domains includes resource constraints.

We have shown that the RTS can implement tactical, run-time performance tradeoffs through both if-time TAPs and planned, interruptible behaviors. If-time TAPs allow CIRCA to take advantage of slack time that only becomes available at runtime. Planned behavior tradeoffs, on the other hand, are reflected in the world model, and are thus subject to the introspective reasoning of CIRCA's AIS.

We have also demonstrated several different strategic tradeoff methods that the AIS can use to modify its plans. Table 7.1 presents a summary of the various strategic tradeoff methods we have investigated, showing briefly how each method can affect several important measures of CIRCA's performance:

Schedulability: The ease with which the Scheduler can build a TAP schedule meeting all timing constraints. The application of tradeoff methods is generally motivated by an inability to schedule the desired TAPs for a particular domain.

Planning completeness: Does the AIS consider all possible world model states when generating the TAP plan? Clearly, if planning completeness is sacrificed, reaction completeness is also.

Reaction completeness: Does the TAP plan specify reactions for all of the states that require action? Together, reaction completeness and reaction confidence lead to performance guarantees: if a TAP schedule can be produced that is fully complete and confident, CIRCA guarantees to prevent failures.

Reaction confidence: How certain can we be that the TAP plan will indicate the appropriate reactions?

Reaction quality: A domain-dependent measure of how well the system's reactions deal with the environment. Example quality measures include precision, accuracy, utility, etc. In the Puma domain, for example, quality might be the number of parts packed into a single box.

The explicit tradeoffs charted in Table 7.1 are a critical feature of the CIRCA approach, allowing the AIS to dynamically, gracefully degrade the guaranteed performance it demands from the RTS. Furthermore, the system has the ability to project the results of these tradeoff methods, as described above. Thus the AIS can decide whether a tradeoff will lead to acceptable performance or not, and it can use this information to guide a search for the best tradeoff to choose.

Tradeoff method	Schedulability	Planning Completeness	Reaction Completeness	Reaction Confidence	Reaction Quality
Ignore a temporal transition	$\Uparrow$	$\Downarrow$	$\Downarrow$		
Ignore an event transition	$\Uparrow$	$\Downarrow$	$\Downarrow$		
Modify a temporal transition	$\Uparrow$				$\Downarrow$
Method selection	$\Uparrow$				$\Downarrow$
Remove a disj. TAP test	$\Uparrow$		$\Downarrow$	$\Downarrow$	$\Downarrow$
Remove a conj. TAP test	$\Uparrow$			$\Downarrow$	$\Downarrow$

Table 7.1: Comparing the effects of various strategic tradeoff methods. Arrows indicate the relative increase or decrease in the performance characteristics.

CHAPTER 8

CONCLUSION

This dissertation began with the goal of merging real-time computing and AI methods. In pursuit of that goal, we first developed a clear view of the various approaches to real-time AI and their differing objectives. To facilitate this understanding, we introduced the any-dimension algorithm concept in Chapter 2. This conceptual framework was used to classify approaches to making performance guarantees, and was also used to understand the immutable laws limiting the use of iterative improvement methods in pursuit of real-time AI (namely, that simple conjunctions of quality-based and resource-based termination conditions are not effective). The conclusion: purely tactical, on-line methods of making guarantees, such as any-time algorithms, are not sufficient to provide the guarantees of timeliness and result quality required by hard real-time domains.

In Chapter 3 we presented a review of previous work categorized into three main areas: AI systems embedded in real-time domains, real-time reactions embedded in AI systems, and cooperative real-time and AI systems. Our research on CIRCA epitomizes the final approach, having goals somewhat different than the first two approaches. CIRCA does not attempt to force its AI methods to meet deadlines; instead, it focuses on isolating its AI methods from the domain-imposed deadlines, while still maintaining clear lines of communication and control between the AI subsystem and the real-time subsystem. CIRCA combines aspects of both strategic and tactical methods to build and execute guaranteed reactive plans. As such, a major contribution of this work has been the delineation of the CIRCA subsystem responsibilities, and the way in which those subsystems cooperatively implement both performance guarantees and unpredictable AI algorithms.

Chapter 4 presented the world model that CIRCA uses to build real-time control plans. This model forms the basis for CIRCA's fundamental conceptual contribution, the notion that the system's AI methods can be isolated from the domain deadlines by using a set of safety-maintaining control reactions. The characterization of a safely-controlled set of world model states represents a unique transfer of the control-theoretic concept of stability to the classical discrete AI planning model.

In addition to supporting the isolation of real-time and AI in CIRCA, the world model also allows the system to implement efficient reasoning about the worst-case possible behaviors of the environment. The world model's minimal representation of time and uncertainty provides a sound basis for the derivation of reactive TAP schedules that can enforce guarantees of safety. As such, CIRCA represents a notable contribution because it *internally* reasons about and modifies its set of performance guarantees, as opposed to previous systems having fixed sets of guarantees derived *externally*.

CIRCA thus has a general capability to "consciously" trade off all of its dimensions of resource usage and performance. The architecture and the graph model provide a unified framework for reasoning about focusing sensing, restricting deliberation, and modifying performance parameters such as timeliness, confidence, and precision. This represents a significant advance in flexibility over previous systems tailored to reason about a limited set of performance or resource dimensions.

Our research on CIRCA has not been focused solely on the theoretical basis for the system's guarantees. We have also developed a prototype implementation of CIRCA, exploring the practical issues in meeting the demands of the architectural design. We have described the development of several unique and powerful software systems that, acting in concert, address the full range of problems involved in intelligent real-time control. These mechanisms include:

- **The AIS interpreter**, which combines the ability to run arbitrary Lisp code with meta-level reasoning, timeouts, and interrupt-based communication. These facilities make our AIS well-suited to deliberation about real-time systems, because the system can not only run classical AI planning algorithms, it can also introspect on its own performance and that of the real-time subsystem it is guiding. This allows the AIS to provide both deliberation and deliberation scheduling, both long-term lookahead planning and alertness to environmental changes.
- **The TAP planner module**, which uses modular decision functions to build reactive control plans based on the world model. The planner implements a simplified temporal logic useful for planning preemptive reactions, and incorporates other special features such as the ability to reason about nondeterministic transitions. The decision function form makes extensions to the planner straightforward, and the planner's explicit stack allows it to be interrupted and resumed without great cost.
- **The Scheduler module**, which implements algorithms to efficiently produce cyclic schedules of reactions selected by the AIS. By reasoning explicitly about the timing behavior of the RTS, the Scheduler sets CIRCA apart from most other AI-based systems, which have no capacity to guarantee timely responses in worst-case circumstances.
- **The RTS**, which provides completely predictable execution of TAPs by using polling, bounded communication primitives, and a low-overhead context-switch scheme. The RTS also executes if-time TAPs to utilize scheduled time that becomes available at runtime.

The prototype CIRCA implementation has been equipped with several methods for making performance tradeoffs, as discussed in Chapter 7. Experiments have demonstrated a wide range of performance tradeoffs that the system can implement in a self-aware fashion, recognizing the resulting changes in resource requirements and output quality.

In sum, we have successfully designed, implemented, and tested the CIRCA approach to combining real-time and AI methods. The current implementation is fairly complex, and has not been carefully optimized. However, the system has been applied to several domains, and has demonstrated its unique combination of AI methods with guaranteed real-time reactive plans.

8.1 Future Directions

Many research areas remain open for future expansion of this work on CIRCA, and on real-time AI in general. The following topics seem well-suited to immediate research and development:

- **Predictive sufficiency.** In Section 4.12 we introduced the notion of predictive sufficiency. Currently, CIRCA does not implement the testing necessary to make sure that conditions of predictive sufficiency hold; the system designer is still responsible for those details. However, there is potential to automate this task as well, and allow explicit reasoning about predictive sufficiency to have impact on the planner's decisions. For example, the system might recognize that it does not have the ability to avoid inappropriate actions from a particular world state (see Section 4.12), and thus it should pursue a different plan. In Appendix D we discuss preliminary ideas on adding this capability to CIRCA, and illustrate the application of those ideas to avoiding inappropriate actions and also to automating decisions about caching sensor data.
- **TAP improvements.** The Scheduler currently builds schedules of individual TAPs, which can lead to rather inefficient behavior, particularly when different TAPs share many feature tests. While efforts to improve the caching of sensed data can help combat this inefficiency, as discussed in Appendix D, this approach will not solve the underlying problem. Much of the time, individual TAPs are not applicable, and their scheduled worst-case execution time is filled in by if-time TAPs. An alternative approach would be to group together TAPs into larger *composite* TAPs that could share the results of various sensing actions and be scheduled as a mutually-exclusive group, requiring less time on the schedule and leading to higher average utilization of that scheduled time. Appendix C discusses more details on the composite TAP approach, giving examples showing when this approach has advantages, and when it does not.
- **Scheduler feedback improvements.** In the prototype implementation, the Scheduler returns either a successful schedule or `nil`. We plan to investigate ways in which the Scheduler can provide more informative feedback about the cause of a scheduling failure, so that the AIS can make intelligent decisions about how to modify the TAPs, the system goals, or the world model. For example, the Scheduler might indicate which TAP timing constraints or resource requirements were most restrictive and prohibited a successful schedule. This research has the potential to move towards a new view of scheduling as an iterative negotiation process requiring feedback.
- **Rational tradeoff motivations.** Currently, the AIS relies on human advice to decide which of its several tradeoff methods should be used in a particular situation.

There is obviously potential to develop more rigorous, automated methods for making these decisions. Decision theory holds promise as a principled method for choosing between alternative tradeoffs, using the concept of expected utility. To employ decision theoretic methods, representations of probabilities and payoffs must be added to the world model. The payoff information might take the form of a more general set of goal priorities, as opposed to the binary priorities currently available (critical and not).

- **Multi-agent considerations.** The world model shows how the AIS reasons about the control-level deadlines it must guarantee through the RTS. From one perspective, control-level deadlines can be seen as the results of commitments with the environment (e.g., starting forward motion commits a mobile robot to an obstacle detection behavior). Similarly, many task-level, non-critical, or "soft" deadlines can be viewed as resulting from commitments with other agents. For example, a mobile robot's deadline of reaching a landmark by a certain time might be based on an obligation to rendezvous with another vehicle. In the Puma domain, the robot arm's desire to pack parts into boxes stems from its commitments to other agents on the assembly line.

  These task-level deadlines have no intrinsic source; they only result from commitments (implicit or explicit) with other agents. Thus it may be possible to avoid or ameliorate violations of such task-level deadlines by re-negotiating the source commitment, in order to alter the associated deadline. This would provide an additional method to avoid task-level failure, supplementing the tradeoff approaches discussed in Chapter 7. Investigating this view of task-level deadlines will require examining issues of modeling other agents' goals and plans, as well as commitments and negotiation with those agents. The possibility of negotiating new task-level deadlines also reveals tradeoffs between spending time negotiating and altering performance to meet existing deadlines.

APPENDICES

APPENDIX A
THE PUMA SIMULATOR

The Puma simulator runs within the Deneb Robotics Igrip simulation environment, and incorporates a number of unusual features required by that system. Igrip was selected as a basis for the simulator because it provides built-in graphical display capabilities, CAD-like three-dimensional object design, simulation of linear and angular joint motion, and even an existing Puma robot model complete with inverse kinematics. Given these existing features, developing the simulation domain for this thesis should have posed little difficulty. As will be seen, however, various limitations in the Igrip system have made the actual development quite time-consuming, and the resulting product is less-than-ideal.

To interface the RTS with the Igrip simulation (and also to the real-world Hero robot), Unix sockets were used to communicate robot and sensor commands, as well as their results. The simulator thus includes a command interpreter which parses these incoming RTS commands and executes the appropriate simulation routines. The interpreter is written in Igrip's Graphical Simulation Language (GSL), which allows the user to specify a device's movements, as well as controlling communication and simulated sensing. Much of the complexity of the simulation resides in this Puma command interpreter, as will be described below.

The conveyor belt and arriving parts are simulated in a simple but deceptive way: the belt itself is a static device that does not move, while the parts arriving on the belt are actually programmed devices moving themselves. Each part runs the same GSL program, which varies its behavior based on the unique name of the actual part running the program. The **part.gsl** program simply looks up the specific part's name in a statically-defined list, finds the associated starting position programmed by the user, and moves the part to that position when the simulation begins. It then issues a single motion command directing the part to move at a fixed rate towards the end of the conveyor— the result is a sequence of parts "marching in step" down the belt. The user can easily specify fixed spacings between the arriving parts, or add in programmed random disturbances.

The emergency alert light is also implemented by a simple GSL program which repeatedly delays for some amount of time determined by a user-controlled random number generator, and then turns on the alert light and sets a global variable indicating an emergency is present. This global variable is accessible to all programs running in the simulation environment. When the Puma interpreter completes the process of pushing the button, it also resets that global variable and turns off the graphical alert light display. An emergency failure is considered to have occurred if the global variable is already set when the alert program tries to set it (i.e., the Puma did not respond to the last emergency before the next one arrives).

Simulated robot sensors have been implemented in several forms. Initially, the sensor that detects when a part is available on the end of the conveyor was implemented using an Igrip ray-casting primitive triggered by the Puma interpreter. An invisible ray was extended out from the end of the conveyor, and if its intersection with the nearest part on the conveyor was close enough, the part was considered reachable. This process proved quite time-consuming, so a much simpler alternative is now in use. When a part reaches the end of the conveyor, the move command issued by its instantiation of **part.gsl** returns, and the part simply sets a global **part-on-conveyor** variable indicating it has arrived. The puma interpreter resets this global after picking up a part. This simple approach also allows the system to automatically keep track of the number of parts that arrive and the number that "fall" off the end of the conveyor; if the **part-on-conveyor** global is already set when another part arrives, then the part is considered to have fallen off, and a global counter is incremented. At the end of a simulation run, the Puma interpreter can print out or return to the RTS the statistics on emergency alerts and part arrivals, and the associated failures.

Similar approaches have been used to implement detection of the arriving part's shape. Rather than simulate a realistic sensor, information on part shape is made available internally by arriving parts, and the Puma interpreter accesses those internal representations to respond to RTS queries about part shape. While these simplifications make the simulated sensors unrealistically reliable, they were useful both for achieving sufficient simulation speed and for getting the Igrip simulation running with a reasonable amount of effort. In the next section, we describe the main issues which made the simulator development non-trivial.

Programming Challenges

Two issues are at the center of most of the interesting Igrip simulator development problems. The first is Igrip's inability to interrupt ongoing robot motion; once the simulator is given a command to move the robot's joints by a certain amount, it will not return control to the interpreter until that process has finished. This is a problem because the CIRCA model of the Puma domain sensibly includes the possibility of interrupting ongoing, long-term motion processes if the environment dictates that a change of course is necessary. For example, the movement of the Puma over the box is modeled by a temporal transition, which may take an indeterminate amount of time. Thus the system must also be able to follow the **stop-moving** action transition, so that it can halt and respond to emergency alerts that may arise during the process of moving over the box.

Implementing interruptible motion required convoluted programming using an approach originally designed and implemented by Mike Hucka. Essentially, the approach is to split up robot motion commands into very small movements that can be completed within a reasonably small time, which is then taken to be the smallest unit cycle time of the simulation (and thus the minimum time before an interrupt can be processed). In the Puma simulation, each motion command sent to the interpreter is saved into registers that record the number of degrees of motion desired for each of the robot's six joints. On each cycle of the interpreter (every .05 seconds), each joint with a non-zero degrees-to-go register is moved by a small increment computed from the joint's speed assignment and the cycle time. This rapid, incremental motion yields a relatively smooth-moving graphical display, largely because of the very fast graphics processing of the Silicon Graphics computer it runs on. Ongoing motion commands are easily interrupted by simply resetting the degrees-to-go registers to new values (or zero).

The second main problem with the Igrip implementation is more serious, and not easily solved. Essentially, the timing primitives supplied by the Igrip environment are not accurate (or even consistent), and thus adjusting the timing behavior of the simulator is more an art than an exact science. Timing in Igrip can be specified in several ways; experience has shown that perhaps the best way is to simply specify a total time requirement for each robot or part motion command sent to the Igrip environment. Thus each incremental joint movement of the Puma is specified to take a certain number of milliseconds, and likewise the motion of parts down the conveyor is assigned a total required time. Unfortunately, while these timing values are usually kept relatively consistent (i.e., a three-second motion does take approximately three times longer than a one-second motion), they are not strongly tied to wall-clock time. That is, a motion command that is assigned for three seconds may take anywhere from three to fifteen seconds to simulate, depending on the other loads on the simulator. Igrip does not seem to make strong efforts to link real-world time with simulation time. While this is fine for an isolated simulation (and probably even desirable, so that simulations can be run faster than real-time), it is a problem when the simulation is interacting with an external process like the RTS, which is attempting to make reliable guarantees on timely behaviors in the simulation.

Extensive experimentation revealed no way to solve this problem directly, so the experiments described in this thesis using the Puma simulator have been hand-calibrated. The actual speed of the simulation depends on the number of parts being simulated on the conveyor, as well as the rate of sensor accesses and other RTS activity. Thus part arrivals and alert arrivals were adjusted for each specific experiment to achieve common rates, within the relatively loose bounds possible given the Igrip limitations.

All these problems with the simulation environment should be taken into account in future plans to develop simulations using Igrip— in general, this system seems too slow and inconsistent to be fully effective. While the easy access to graphics and inverse kinematics are a huge bonus, the lack of timing control available to the user makes the final simulation system almost as difficult to work with as a real-world robot.

APPENDIX B

COMMUNICATING WITH THE RTS

The TAP schedules executed by the RTS are sent to it by the AIS over a socket, in a special language form described by the grammar shown in Figure B.1. The download begins with the material defining each of the TAPs that will be used in the schedule. Each TAP is made up of a test section, defining the set of AND/OR combinations of primitives tests that should be executed, and an action section, listing the primitive actions that should be performed if the tests return true[1]. Tests are described by the name of a primitive testing function that should be executed to yield some result value describing the current state of the world, and a testing value against which that result value should be compared. Figure B.2 illustrates an example TAP schedule download message.

As each TAP is read in by the RTS, it is assigned a unique integer index corresponding to its position in the list of TAPs. These indices are then used to communicate the actual schedule of TAP executions, following the BEGIN-SCHEDULE keyword. The schedule is thus a list of TAP indices, in the order in which they should be executed. Because of differing periods and the effects of the Scheduler, this list may contain many repetitions of each TAP index. This observation is the motivation for using this index method for communicating the TAP schedule: the initial TAP definitions are used to form data structures on the RTS, which are then easily and efficiently referenced by the indices, rather than communicating multiple definitions of the same TAP for each of its invocations in the schedule.

The looping to be performed at the end of the TAP schedule is implicit at the end of the list of TAP indices, tagged by the END-SCHEDULE keyword. The list of if-time TAPs is communicated in a similar fashion, except that in this case the order of the TAP indices has no significance, and no repetition will be included, because the RTS dynamically decides when each if-time TAP will be executed. The `#` acts as a message terminator, letting the RTS know that it has received the entire TAP schedule download and should begin processing the schedule and making it ready for execution. Currently, the RTS does incrementally download the new schedule, but it cannot parse/process the new schedule until it has all been received. So the `#` triggers the parsing processing, whose runtime actually depends on the length of the new schedule, but is very short in any case. A more complex re-implementation in `flex` (rather than `lex`) would allow incremental parsing.

[1]Currently, the AIS only plans one action per TAP, and the RTS can only parse one action per TAP. Extensions to the RTS parser for multiple actions are trivial.

⟨schedule⟩ →
    ⟨tap-list⟩ **BEGIN-SCHEDULE** ⟨tap-index-list⟩
    END-SCHEDULE ⟨if-time-section⟩ #

⟨if-time-section⟩ →
    BEGIN-IFTIME ⟨tap-index-list⟩ **END-IFTIME** |
    ϵ

⟨tap-list⟩ → ⟨tap⟩ ⟨tap-list⟩ | ⟨tap⟩

⟨tap⟩ →
    BEGIN-TAP ⟨test-expr⟩ **ACTION** ⟨action-primitive⟩ **END-TAP** |
    BEGIN-COMPOSITE ⟨tap-index-list⟩ **END-COMPOSITE**

⟨test-expr⟩ →
    ⟨test⟩ |
    (NOT ⟨test-expr⟩ **)** |
    (AND ⟨test-expr-list⟩ **)** |
    (OR ⟨test-expr-list⟩ **)**

⟨test-expr-list⟩ → ⟨test⟩ ⟨test-expr-list⟩ | ⟨test⟩

⟨test⟩ → **(** ⟨test-primitive⟩ ⟨value⟩ **)**

⟨test-primitive⟩ → ⟨identifier⟩

⟨action-primitive⟩ → ⟨identifier⟩

⟨value⟩ → ⟨identifier⟩

⟨identifier⟩ → ⟨alpha⟩ | ⟨identifier⟩ ⟨alpha⟩ | ⟨identifier⟩ ⟨digit⟩

⟨tap-index-list⟩ → ⟨tap-index⟩ ⟨tap-index-list⟩ | ⟨tap-index⟩

⟨tap-index⟩ → ⟨integer⟩

⟨integer⟩ → ⟨digit⟩ ⟨integer⟩ | ⟨digit⟩

⟨alpha⟩ → **A**|**B**|**C**|**D**|**E**|**F**|**G**|**H**|**I**|**J**|**K**|**L**|**M**|**N**|**O**|**P**|**Q**|**R**|**S**|**T**|**U**|**V**|**W**|**X**|**Y**|**Z**|**_**|**-**

⟨digit⟩ → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Figure B.1: A pseudo-BNF grammar for downloading TAP schedules to the RTS. Non-terminals are bracketed by ⟨ ⟩ and alternatives are separated by |.

```
BEGIN-TAP (BOX1_BOUNCED NIL) ACTION BOUNCE_BOX1 END-TAP
BEGIN-TAP (CURSOR_MOVED_IN_WINDOW T) ACTION MARK_CURSOR END-TAP
BEGIN-TAP (AND (CURSOR_MOVED_IN_WINDOW NIL)
               (AND (BOX1_BOUNCED T) (BOX2_BOUNCED NIL)))
          ACTION BOUNCE_BOX2 END-TAP
BEGIN-SCHEDULE  0 1 END-SCHEDULE
BEGIN-IFTIME 2 END-IFTIME #
```

Figure B.2: A simple TAP schedule download from the AIS to the RTS, for the bouncing box domain.

Sharing Identifiers

The grammar of Figure B.1 shows that primitive tests, comparison values, and primitive actions are communicated to the RTS using standard alphanumeric identifiers. However, because the RTS is written in C and the AIS is written in Lisp (and runs on a different processor), the programs must have some additional mechanisms to ensure that the meaning of these identifiers is "common knowledge." In other words, when the AIS sends a TAP to the RTS invoking the **push-emergency-button** action, the RTS must be sure to bind that identifier to its routine that performs the appropriate action.

Several fancy programming tricks are used to make sure that this binding process is done correctly, with very little effort by the system designer. The goal of these tricks is to make sure that an identifier used in the Lisp description of a domain (from the file "trans.lisp") is bound to an RTS primitive (from the file "primitives.c") with the same name. To get the naming to match up, we use a `perl` program that parses both "trans.lisp" and "primitives.c," and produces a series of files that will enable the appropriate mapping.

The `perl` program, `parse-trans`, makes the RTS implement this mapping by actually defining (before compile-time) part of the lexical analyzer used by the RTS to scan incoming messages from the AIS. `Parse-trans` reads the "trans.lisp" file and extracts the names of tests, values, and actions used in the world model description. It then reads the "primitives.c" file and extracts the names of defined C primitive functions. A mapping file "primitives.h" is generated to automatically map the Lisp test and action names to indices (unique integers). The file "primdefs.c" is also generated, to initialize (at run time) an array of function pointers indexed by the indices defined in "primitives.h." Then `parse-trans` builds a partial `lex` source file "auto.lex" that defines the necessary lexical analysis machinery to parse the Lisp primitive names and interpret them as their respective indices, which can then be used by the RTS interpreter mechanism.

So, in summary, the RTS' `lex`-generated parser will translate incoming Lisp test and action names (identifiers) into indices, which index the automatically-generated array of C function pointers. To execute downloaded TAP tests and actions, then, is merely a matter of de-referencing the corresponding function pointers.

For further clarification, we now present a short example. Suppose the Lisp file "trans.lisp" has a transition defined as:

```
;; in trans.lisp, user generated
(my-make-instance 'temporal
        :name "emergency-failure"
        :preconds '((emergency T))
        :postconds '((failure T)))
```

Then `parse-trans` will recognize "emergency" as a test primitive and build a line in "primitives.h" that gives that name an index:

```
/* in primitives.h, auto generated */
\#define EMERGENCY 24
```

Then `parse-trans` will look for some C code in "primitives.c" defining a corresponding primitive function:

```
/* in primitives.c, user generated */
int emergency ()       /* TIMING: 400 */
    { ... }
```

Finding that, `parse-trans` will make entries in "primdefs.c" that make that primitive accessible (note that the timing information is made available, for possible use by the RTS):

```
/* in primdefs.c, auto generated */
primitives[EMERGENCY] = emergency;
wcet[EMERGENCY] = 400;
```

And finally, the entry in "auto.lex" will allow the RTS to parse incoming TAPs from the AIS that include the string "EMERGENCY":

```
< in auto.lex, auto generated >
EMERGENCY       parse-prim(24);
```

The main advantage of `parse-trans` is that it allows the user to make additions to the set of available primitives without ever hand-modifying the lexical analyzer or the `#define`s mapping names to indices, etc. Adding a new primitive is only a matter of defining the C code and using it in the Lisp world model transitions, rather than also making all those other code modification, which would present too many chances for errors or forgetting to add one new entry to some file.

APPENDIX C

COMPOSITE TAPS

Consider three TAPs A, B, and C, scheduled to run sequentially, as shown in Figure C.1a. Suppose that these TAPs have been planned to respond sequentially to some domain event, detected initially by TAP A. After TAP A detects and responds, the plan calls for TAPs B and C to respond, at which time the response will be complete. This type of situation arises in the Puma domain, for example, when the robot is moving a part to the box and the emergency light activates: a TAP detects the situation and stops the motion, another TAP puts the part on the table, and a third TAP finally makes the robot push the button to cancel the emergency.

The casual observer might have the impression that, in the worst case, the response time guarantee that could be made for the completion of that sequence of responses would be $tests(A) + tests(B) + tests(C) + P$ (where $P = wcet(A) + wcet(B) + wcet(C)$), as shown in the hypothetical timeline of Figure C.1b. Unfortunately, this intuitively desirable derivation is not correct.

The first problem with the timeline is that it ignores the effects of if-time TAPs: when if-time TAPs are available, the RTS will execute them as many times as possible to fill unused scheduled time. Thus the first invocation of TAP A in the timeline will not use up just its test time: the if-time TAPs may use up the action time as well. Making this correction to the timeline, as shown in Figure C.1c, we might then postulate that the appropriate response guarantee would be $2 * P$. However, this response deadline cannot be derived by the current Planner and Scheduler, and is not necessarily correct.

The remaining problem is that this formulation makes assumptions about dependencies between the TAPs. The Figure C.1c timeline assumes that TAP B will execute its actions on its second invocation because TAP A was fired immediately before then. However, the Scheduler has no knowledge of the dependencies we have postulated between these TAPs, so it has no way of knowing that, because TAP A fired, we expect the preconditions of TAP B to hold. Furthermore, there are many cases where sequentially ordered TAPs would not actually fire immediately following one another. In the above Puma example, suppose the TAP that halts the robot before putting the part on the table does so by sending a command which may have slightly delayed effects, so that the robot may still be moving for a few milliseconds after the TAP has completed. In that case, if the preconditions of the subsequent **place-part-on-table** TAP require the robot to be not moving, then the test may return false, and thus TAP B might not fire immediately. In that case, the RTS will

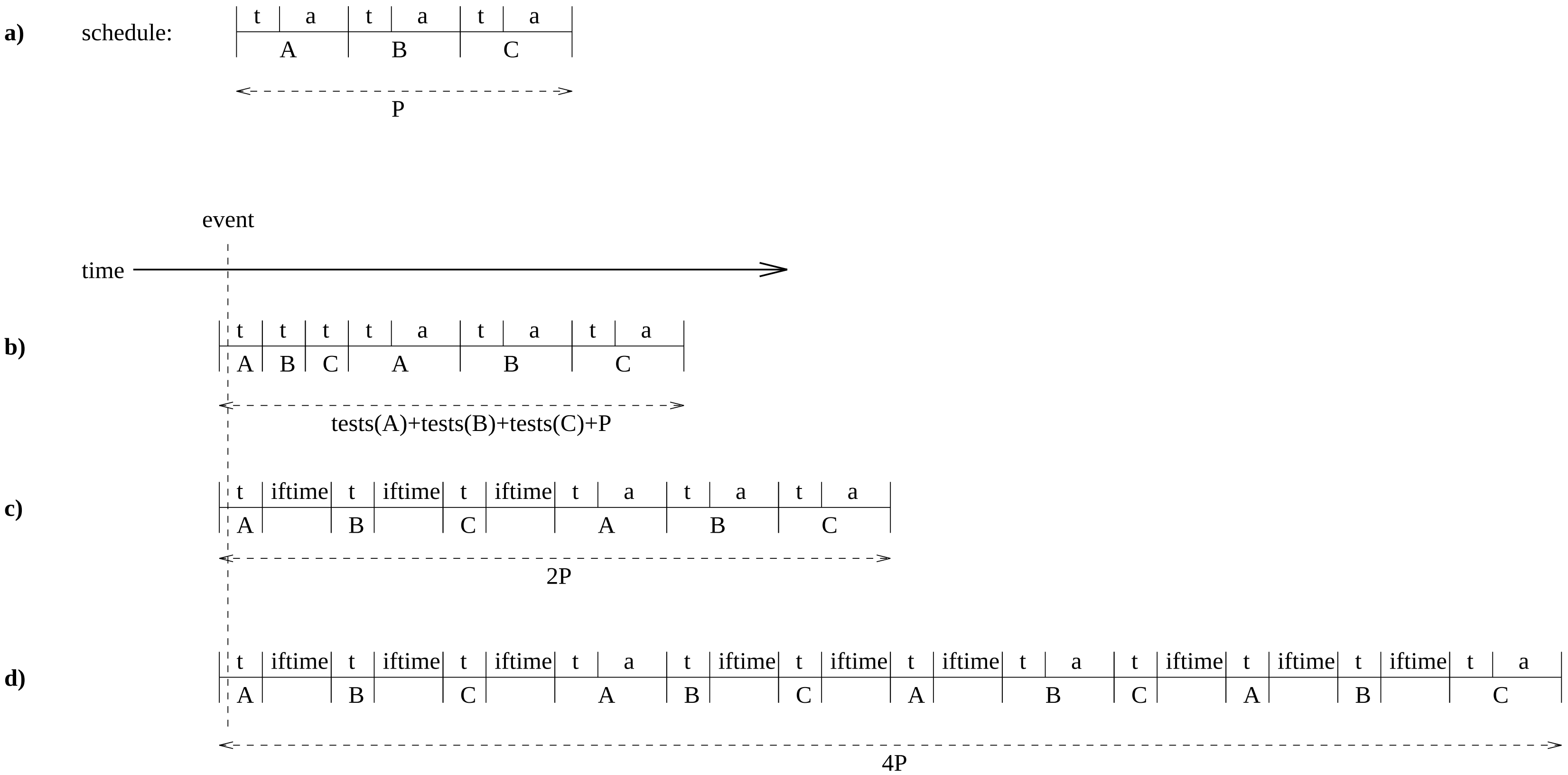


Figure C.1: Deriving the response time for a sequence of TAPs.

have to finish another cycle of the schedule before the sequence of TAP firings continues. As illustrated in Figure C.1d, this means that each TAP in the schedule actually has a worst-case possible response time of $wcet(\tau) + P$, as derived earlier (and despite any sequential constraints we might derive from the plan). The effect of this equation, in the case of sequential TAPs, is to add a full TAP cycle period to the guaranteed response time for each TAP in the sequence. In general, for a sequence of TAPs $\tau_1, \tau_2, ...\tau_n$, the best guarantee that can be made for the overall sequence response time is

$$\sum_{i=1}^{n} (wcet(\tau_i) + P(\tau_i))$$

Thus the best guarantee that can be made for the A–B–C sequence of this example is $4 * P$.

This bound on response time is caused by the individual scheduling of TAPs, so that regardless of how frequently they are applicable, their full worst-case execution time is always scheduled, and if-time TAPs will use that time if the TAP tests fail. For sequences of TAPs, it should be clear that this leads to rather inefficient and frequently unacceptable response-time results. For example, consider the Puma domain example above, in which three TAPs must fire in a sequence to respond to the emergency alert. Suppose, for simplicity, that each of those TAPs requires three seconds to execute, in the worst case (this includes the time for robot motion, so it is not a very unrealistic value). With only those three TAPs in the schedule, the invocation period for each TAP would be 9 seconds. Thus the best response that could be guaranteed for the sequence of three TAPs would be 36 seconds, despite the fact that only 9 seconds of actual TAP execution is required.

Motivated by this observation, we have investigated a method for building *composite* TAPs out of individual TAPs, combining their functions in search of greater efficiency.

Building Composite TAPs

The construction of composite TAPs is illustrated in Figure C.2. In Figure C.2a, the three TAPs from the previous example have been combined into a single TAP which acts like a Lisp **cond** construct, sequentially running the tests of each TAP until one returns true, and then executing the appropriate actions. If none of the TAPs' tests return true, the composite TAP completes, and the RTS may fill in the unused scheduled time with if-time TAPs, as usual.

The worst-case execution time of a composite TAP is determined by the worst-case path through its component TAPs. In Figure C.2a, the worst-case path involves testing for TAPs A and B but not executing their actions, and then testing TAP C and executing its actions. Of course, the worst-case path may not always result from executing the last component TAP's actions. Figure C.2b shows a composite TAP in which the worst-case path involves executing the second TAP; all other execution paths are shorter, even those involving execution of a later component TAP's actions.

The max-period of a composite TAP C is derived to ensure that each of its component TAPs $\tau_1, \tau_2, ...\tau_n$ is executed frequently enough to guarantee its required $min\Delta$. Accordingly, the following equation is used:

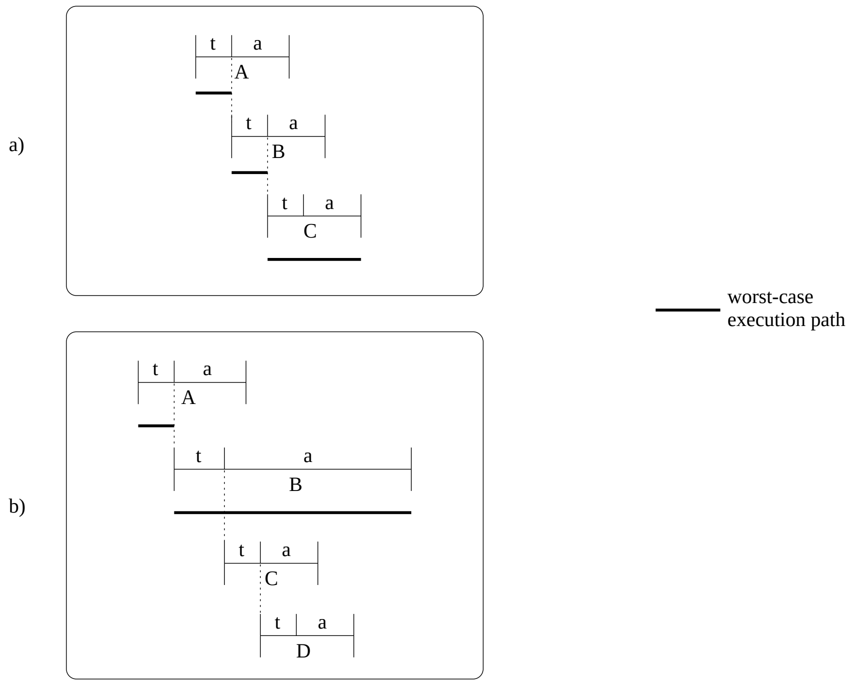


Figure C.2: Two composite TAPs.

$$max - period(C) = \min_{i=1}^{n} (min\Delta(\tau_i) - wcet_i(C))$$

where $wcet_i(C)$ is the worst-case execution time of the composite TAP in the case when the actions of the component TAP τ_i are executed.

A crucial element underlying the success of the composite TAP approach is the fact that individual TAPs always apply to different states: they are never testing for the same state. The testing operations of a composite will never yield more than one appropriate TAP, so no conflict resolution is necessary. As a result, we know that combining TAPs into the composite execution form will never cause a TAP to miss some world state it would otherwise have detected. For example, suppose we have a TAP T that the planner has determined should be run at least once every second, in order to detect some state S and avoid imminent failure. If we combine T into a composite TAP, the above constraints ensure that the composite TAP will be executed at least once every second. We know that the state S must persist for at least one second before any reaction must occur, because the planner based the period of the original T on that information. Therefore, since no other TAPs will fire for the given state, the composite that includes T will detect and react to the state, just as the original version did.

The advantage of composite TAPs is that they avoid scheduling TAPs individually, and thus they avoid having large quantities of schedule time being filled by if-time TAPs. Instead, the composite TAPs provide a more rapid, focused use of RTS time to assess the current world state and invoke the appropriate reaction. Composite TAPs will provide a higher run-time utilization of the RTS for guaranteed reactions, because they apply to more situations than individual TAPs, and less scheduled time is filled by if-time TAPs.

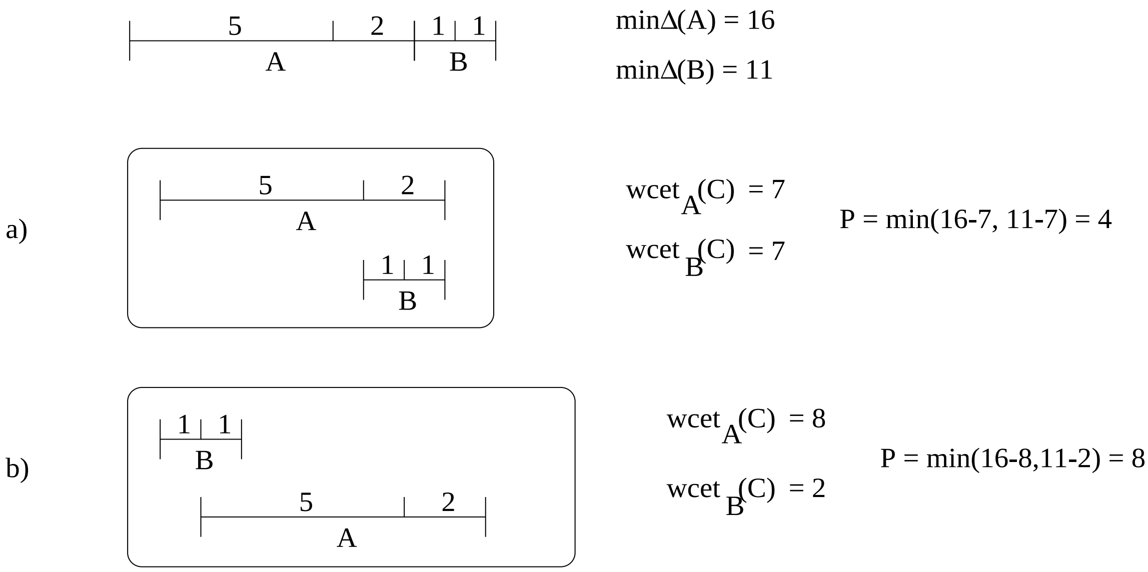


Figure C.3: Example TAPs showing that ordering in a composite is significant.

The main disadvantage of composite TAPs is that they add to the worst-case execution time of TAPs that are later in the ordered composite list. As a result, choosing which TAPs to include in a composite can be important, and the ordering of the component TAPs is also very important. Combining TAPs with time-consuming tests can make it difficult or impossible to schedule the resulting composite, even if the individual TAPs can be scheduled separately. Two trivial examples will serve to illustrate these problems.

The example in Figure C.3 shows two TAPs which can scheduled to meet their respective $min\Delta$ requirements separately, but cannot be scheduled if they are combined into a composite TAP in the wrong order. In Figure C.3a, the TAP with the longer tests (A) has been placed first, and as a result the worst-case execution time for the composite path which invokes TAP B is much longer than the TAP B alone. The original schedule was able to execute TAP B with a period of 9, and therefore could meet a $min\Delta$ requirement of $P(B) + wcet(B) = 9 + 2 = 11$. To meet this $min\Delta$ with the composite TAP, however, would require a period $P = min\Delta(B) - wcet_B(C) = 11 - 7 = 4$. Since the composite TAP has a worst-case execution time of 7, there is no way it can be scheduled to run with a period of 4. However, Figure C.3b shows the opposite ordering, which is feasible, and actually can achieve a shorter period than the individually scheduled TAPs. In this case, because the costly tests of TAP A are not executed before TAP B, the smaller $min\Delta$ of TAP B can still be met.

However, proper ordering will not necessarily solve these sorts of problems. Figure C.4 shows two TAPs and the two possible composites which can be formed from them. In both cases (a) and (b), the excessive penalty of executing the first TAP's test portion makes the second TAP so delayed that it can no longer meet its $min\Delta$ requirements. In Figure C.4a, for example, TAP B is executed second, and its worst-case execution time is expanded from 6 (in the individual case) to 11 in the composite. Thus to meet its original $min\Delta$ of 19, the composite would need to be executed with a period of 8. But since the composite has

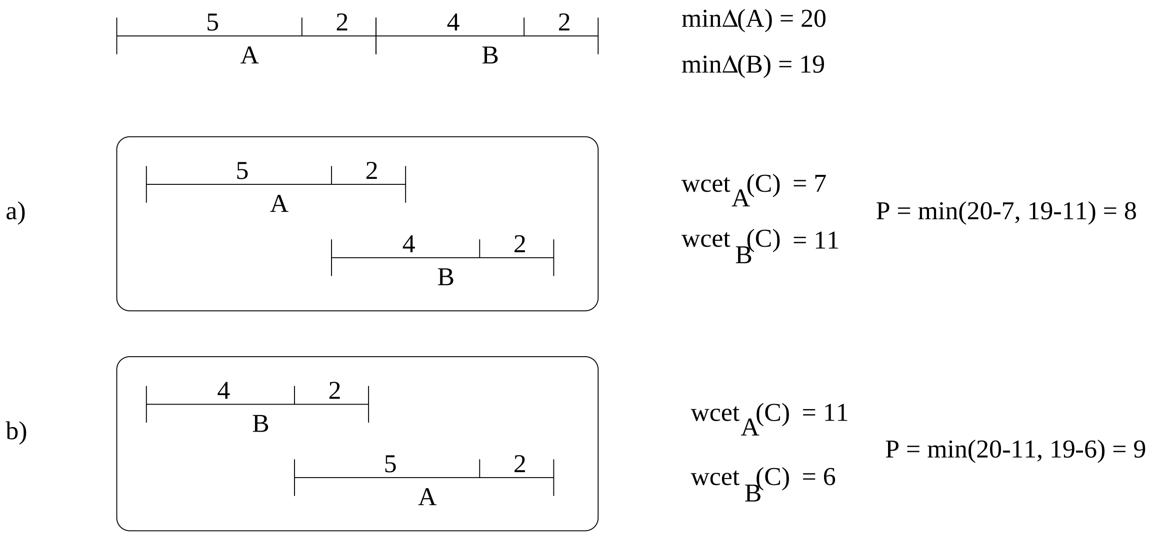


Figure C.4: Example TAPs showing how composites may be completely infeasible regardless of ordering.

a worst-case execution time of 11, this is clearly not possible.

These examples should make it clear that building good composite TAPs is not a simple task: the introduction of interactions between TAP tests and actions essentially makes the construction of composite TAPs a scheduling problem by itself. While we have implemented mechanisms to build, download, and execute composite TAPs, we are still in the process of developing reliable heuristics for selecting which TAPs should be grouped together, and how they should be ordered.

The notion of composite TAPs also points the way towards merging the automatically-generated, scheduled, guaranteed-response-time concepts of TAPs with the form of Universal Plans [66]. Composite TAPs are not particularly efficient because, although they combine multiple TAPs, they leave the test sections of those TAPs discrete and independent. Thus each TAP's tests may repeat checks of state features that have been examined earlier in the same composite TAP (this is not possible in individual TAPs because of the nature of the decision trees ID3 produces). A better approach, as long as we are combining TAPs, would be pass the information about which of the component TAP actions to apply to which states into ID3 to produce a single decision tree with multiple decision values corresponding to which action to execute. The resulting decision tree, resembling a partial Universal Plan for the composite TAP's "states of interest," could be much more efficient than the separate TAP tests, and it would eliminate repetitive tests within the composite TAP.

APPENDIX D

IMPLEMENTING PREDICTIVE SUFFICIENCY

In this Appendix we describe predictive sufficiency in more detail, and present suggestions and ideas about how CIRCA could explicitly reason about and use predictive sufficiency to its advantage.

To accurately describe the concept of predictive sufficiency, we must begin with some notation. We will use a simple temporally-qualified modal logic to describe the state of a control system's knowledge. The logical statement $K(p[t_i], t_j)$ indicates that the system knows, at time t_j, that the proposition p holds at time t_i. For convenience, we will also use statements of the form $K(p[t_\alpha, t_\beta], t_j)$, indicating that the system knows, at time t_j, that p holds continuously over the time interval from t_α to t_β.

A control system's operations can be generally expressed as the acquisition of an observation, the logical deduction of what that observation means about the state of the world at the time the observation was made, the deduction of the predictions that the observation allows the system to make about the world following the observation, and the selection of an action based on that knowledge. In our notation, we have:

$$O[t_i] \quad \stackrel{\text{interpret}}{\longrightarrow} \quad \forall p \in P_i^O : K(p[t_i], t_j)$$

$$\stackrel{\text{predict}}{\longrightarrow} \quad \forall p \in P_p^O : K(p[t_{p\alpha}, t_{p\beta}], t_k)$$

$$\stackrel{\text{select}}{\longrightarrow} \quad a[t_{a\alpha}, t_{a\beta}]$$

where $O[t_i]$ is a sensory observation made at time t_i, P_i^O is the set of propositions which can be inferred about the world at time t_i from the observation, and P_p^O is the set of propositions that can be predicted over the respective intervals $[t_{p\alpha}, t_{p\beta}]$. These intervals are the "intervals of predictive sufficiency," during which the observation O is sufficient to predict the value of the propositions P_p^O. The time t_j is the time by which the system's processing has derived its knowledge of P_i^O, and the time t_k is the time by which the system knows P_p^O. Following those deductions, the action a is chosen and executed during the time interval $[t_{a\alpha}, t_{a\beta}]$.

We use the concept of predictive sufficiency to show how an action can be guaranteed to be appropriate when it is executed. The key to avoiding an inappropriate action is to ensure that the value of the propositions used to choose an action will remain unchanged before and during the action. This can be achieved by making action choices based on propositions whose intervals of predictive sufficiency cover the time during which the action's preconditions are necessary. More formally, suppose the action a requires a set of propositions P_a to hold during the respective intervals $[t_{p_a\alpha}, t_{p_a\beta}]$. If $P_a \subseteq P_p^O$ and $\forall p \in P_a : (t_{p\alpha} \leq t_{p_a\alpha}) \wedge (t_{p\beta} \geq t_{p_a\beta})$, then the intervals of predictive sufficiency that are supported by the observation O ensure that the required propositions will hold as necessary.

For example, consider an intelligent autonomous vehicle that is waiting at an intersection for the traffic signal to turn green. At some point, the controlling agent will make an observation confirming the proposition "the light is green" (P_i^O). This proposition alone is not sufficient to justify crossing the intersection, because there is no guarantee that, at the time t_j when P_i^O is known, the light is *still* green. The knowledge resulting directly from interpreting sensor readings can only describe past states of the world. However, if the system knows some information about the domain's dynamic behavior, it can derive additional propositions that describe the current and future worlds. In this example, the system might know that the traffic signal will switch to yellow for at least five seconds before it turns red. So, although the system does not know if the light is still green, it can conclude that, for at least five seconds after the light was seen to be green, the light must be either green or yellow and the intersection will be "safe" to cross (P_p^O). If the agent is sure that the time it takes to infer these propositions from its observations and cross the intersection is less than five seconds, it can guarantee that it will never be in the intersection during a red light.

Thus the addition of domain modeling information has allowed the system to make explicit predictions about the future state of the world, based on stored sensor readings. Given further information about the agent's own performance, these predictions are then shown to be sufficient to justify certain actions. This example illustrates how predictive sufficiency can cover the sense/act gap, avoiding inappropriate actions.

Avoiding Inappropriate Actions

Figure D.1 shows an example portion of the graph-based world model for the stoplight scenario. The model shows that the stoplight has three main states, Red, Yellow, and Green, corresponding to its signal colors. In the Yellow and Green states, it is safe for the agent to cross ("safe2X"), but not in the Red state. In this simple example we have abstracted out all of the agent's own state except for the indication of whether it has crossed the intersection or not. The different states of the traffic signal are connected by temporal transitions (double arrows) indicating that, as time passes, the signal will transition to subsequent states. Each temporal transition is labeled with the minimum possible delay before the transition occurs, perhaps derived from the agent's previous experience with this traffic signal. For example, the transition between the Red and Green states indicates that the signal will stay red for at least 60 seconds before turning green.

If CIRCA is told that the Red state is its initial condition, it will first try to plan an action for that state. Since the state is not safe for crossing, the only applicable action is

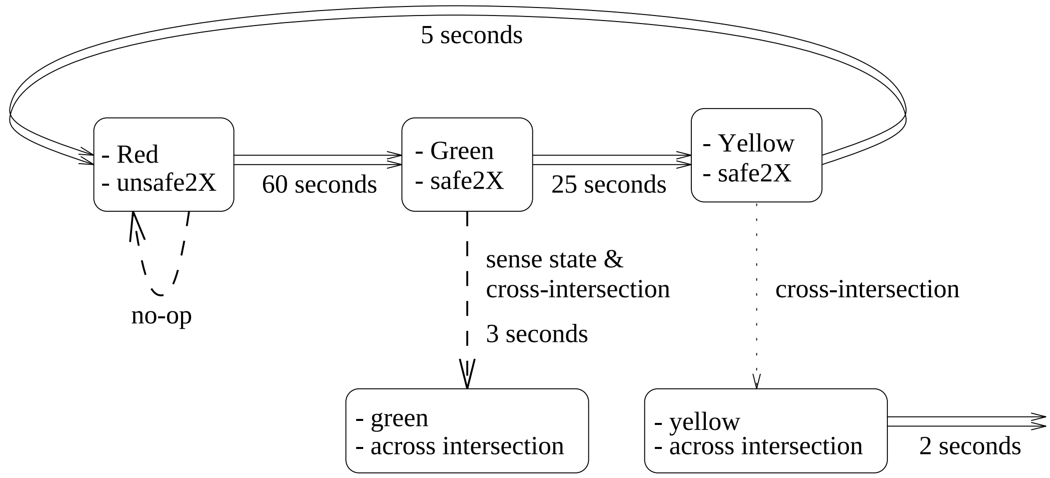


Figure D.1: An abstracted portion of the world model for the stoplight scenario.

`no-op` (shown as a dashed line in Figure D.1). The system then applies its domain rules and derives the temporal transition leading to the new Green state. Again an action is chosen for the new state, but this time the `cross-intersection` action is chosen because it is applicable (Green is safe to cross) and because it leads to the desired result. So at this point CIRCA has planned a simple reaction indicating that, when the light is green, the agent should cross. But the system has not yet shown why this action is guaranteed to be appropriate when executed; it has not yet addressed the sense/act gap, and the possibility that the light will change before the `cross-intersection` action is completed.

CIRCA could explicitly address these issues by ensuring that the propositions used to satisfy the action's preconditions are covered by intervals of predictive sufficiency. The system knows the worst-case execution time of all of its sensing and action primitives, as well as their combinations. Thus the system knows exactly how long it will take, in the worst case, to detect the green light and cross the intersection (here, three seconds). To check for predictive sufficiency, the system should continue its planning and look for other domain processes that may be occurring during the action. In this case, domain knowledge indicates the temporal transition leading from the Green state to the Yellow state after a minimum of 25 seconds.

As noted above, CIRCA does not know how long the light has been green when it is observed; therefore, in the worst case, it should assume that the temporal transition to the Yellow state occurs at the same time the system initiates the transition to cross the intersection. This corresponds to the "ghost" action transition in the figure (the dotted line leaving the Yellow state), showing that the action may actually be applied to the Yellow state, leading to a new state where the signal is yellow, but there is now a minimum of only two seconds before a temporal transition leads to a red light state.

In this process of looking at transitions out of the Green state for which the action is planned, CIRCA has shown that, although alternate results are possible, the precondition of the action ("safe2X") is known to hold for five seconds. This is the interval of predictive sufficiency: seeing a green light allows the system to guarantee at least five more seconds of safe crossing time. Because the process of sensing the green light and then crossing the street takes no more than three seconds, the interval of predictive sufficiency is long enough to cover the sense/act gap. Therefore, CIRCA can plan this action and guarantee that it will only be executed in appropriate situations.

When CIRCA continues the planning process and tries to choose an action for the Yellow state, it finds that the `cross-intersection` action is applicable and leads to the desired state. However, when the system tries to ensure that the "safe2X" precondition can be predicted to hold while the action is executed, it would find that the transition leaving the Yellow state leads to the Red state, which is "unsafe2X." Therefore, since the system does not know how much time may have passed in the Yellow state before the state was detected, and the subsequent state does not satisfy the action's preconditions, the action would be rejected. In summary, CIRCA could use its explicit understanding of predictive sufficiency to derive a common rule of thumb used by drivers who glance at a traffic signal: if the light is green, go ahead and cross; if the light is yellow, do not start crossing, because the light may turn red too soon.

Real-Time Response Guarantees

An interesting feature of this approach to avoiding inappropriate actions is that it requires no information about how frequently a particular sensory observation is being acquired— the example said nothing about how often the system checks to see if the light is green. If the system never even checks to see if the light is green, and thus never takes the `cross-intersection` action, it will never perform an inappropriate action. Clearly, this type of proof is only useful for goals that have no deadline. For real-time goals, that require response-time guarantees, this method is not sufficient.

Suppose we alter the traffic signal domain slightly, so that the system is now required to cross the entire intersection during the first available green light (perhaps because there is an impatient driver behind the autonomous vehicle). This deadline scenario is illustrated in Figure D.2, showing that if the system has not crossed by the end of the green light, it has failed.

CIRCA recognizes this potential failure when it examines the transitions leading out of the Green state, and realizes that it must preempt the temporal transition. That is, CIRCA decides it must execute some action that will definitely occur before the earliest time the temporal transition to failure can occur. To preempt the transition, CIRCA commits to repeatedly executing the behavior that checks for the crossing conditions, at least frequently enough to ensure that the crossing action will be completed before failure can occur.

It is fairly obvious that, to guarantee that it will simply detect the first Green state, which has a minimum possible duration ($min\Delta(P_i^O)$) of 25 seconds, CIRCA must test for the state at least once every 25 seconds. However, detecting the Green state is not sufficient: the system must be able to *finish* crossing before the signal changes to yellow. To provide this predictive sufficiency, CIRCA could rely on its additional knowledge about the frequency with which it will be obtaining sensory information. For example, if the period of the repeated observations is $\rho(O)$ seconds, then an observation in which the condition does hold, following an observation in which the condition does not hold, indicates that the

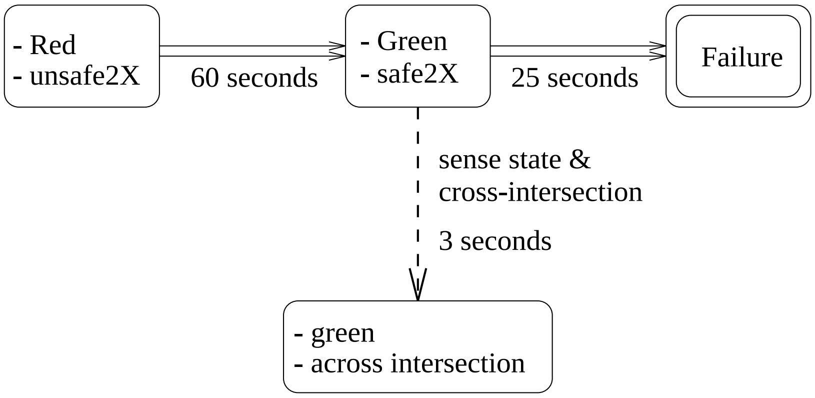


Figure D.2: An abstracted portion of the world model for the modified stoplight scenario with a response-time deadline.

change of state must have occurred in the last $\rho(O)$ seconds. Therefore, the condition must continue to hold for at least $min\Delta(P_i^O) - \rho(O)$ seconds.

Thus we have a modified interval of predictive sufficiency, based on both knowledge of the domain and knowledge about the ongoing performance of the reactive system itself. The AIS could actually reason about the performance of the reactive system it is designing to derive the predictive sufficiency of the observations it plans to make. To guarantee that every real-time reaction will be checked and executed before its corresponding deadline, CIRCA must show that the predictive sufficiency of the observations covers the sense/act gap. That is, $min\Delta(P_i^O) - \rho(O) > t_{a\beta} - t_i$. In our modified traffic signal example, we have $25 - \rho(O) > 3$, so that $\rho(O) < 22$. If CIRCA can guarantee to execute the reaction that tests for Green and crosses at least once every 22 seconds, it can guarantee that it will not fail to cross on the first green light.

Automatically Allocating Internal State

Storing sensory data in internal state is desirable because a system that caches sensory data can access that information many times without incurring the high cost of repeated sensor accesses. However, relying on cached data increases the risk of executing inappropriate actions or missing deadlines, because it increases the sense/act gap. Predictive sufficiency thus plays a role in determining when caching sensory data is acceptable. We are investigating an intriguing approach to automatically planning the use of stored sensory data in the context of CIRCA's RTS.

The first line in Figure D.3 shows an abstract representation of a simple TAP schedule that includes three different TAPs (A, B, and C). CIRCA's planner has determined that TAPs B and C must be run more frequently than TAP A, so the schedule loop includes two invocations of B and C for each invocation of A.

Suppose that both TAP A and TAP B access the same sensor to get information about the same world feature. If accessing that sensor is costly, it may be worthwhile to try to

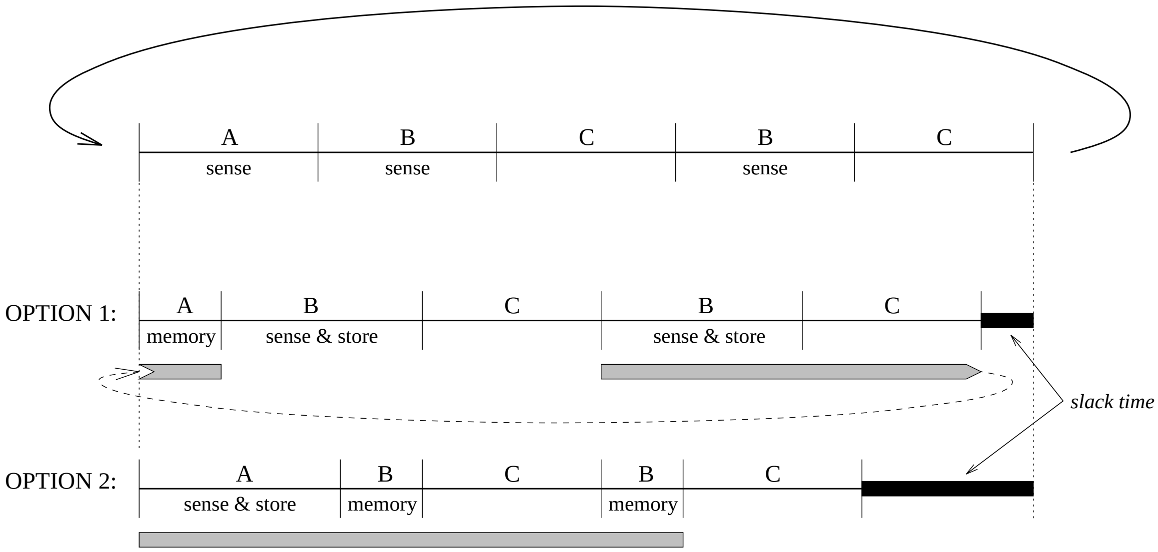


Figure D.3: An example TAP schedule and two modifications that use memory accesses instead of sensor accesses.

minimize sensor accesses, and rely instead on cached sensor readings when possible. To do this, however, we must know that the added delay between when a sensor reading is actually acquired and when it is used (from memory) will not cause problems. This is precisely the knowledge described above: we must know the interval of predictive sufficiency of the sensor reading.

Figure D.3 shows two simple modifications to the existing schedule that might be used to decrease the number of sensor accesses and rely more on memory. In the first modification, the more-frequent TAP B has been modified to store the value that it senses into a memory location[1]. The less-frequent TAP A no longer accesses the sensor, instead relying on the cached values provided by the most recent invocation of TAP B. With these changes, it is clear that TAP A is still using sensor data that is polled at a high-enough frequency: the sensor data is still updated at least once per cycle, so the real-time reaction guarantees discussed in above are preserved. However, there is now a longer gap between the time the data is sensed and when it is used by TAP A. Thus, to avoid failures due to inappropriate actions, the sensor's predictive sufficiency must cover the time from when it is accessed in TAP B through the time TAP A uses the cached value (the shaded region below the OPTION 1 line).

The advantage of making these modifications to the TAP schedule is that, because memory accesses take much less time than sensor accesses, the same set of TAPs will use less time. As a result, CIRCA can use the slack time (the black bar) to either schedule more TAPs or run the same set of TAPs more frequently.

In the second modification option, TAP A stores the result of its sensor access, and the

[1]Because each invocation of a TAP in CIRCA's schedules is a pointer to a single TAP structure, there is currently no way to modify a single invocation. Thus we cannot (yet) have only the second invocation of TAP B save the sensor value.

two invocations of TAP B access that stored value rather than the sensor itself. Clearly this option can provide greater savings than the first, since two sensor accesses have been eliminated rather than one. Figure D.3 reflects this improvement in the longer slack time bar following the OPTION 2 schedule. However, this option has correspondingly more stringent constraints which must be met for the system to remain functionally correct. In the first option, the maximum frequency of sensor accesses for each TAP was not changed, so all the TAPs still could be guaranteed to not miss any conditions they were originally scheduled to detect. In the second option, the sensor is now only accessed once per cycle, so TAP B is no longer working with information updated at the original frequency[2]. Thus, to make sure no transient conditions are missed, the system would have to check that the maximum possible frequency of change for the sensed feature is actually lower than the cycle frequency of the entire TAP schedule, rather than the previous (higher) frequency of TAP B invocations.

Furthermore, as with the first option, the system must ensure that the predictive sufficiency of the data spans each of its uses. In this case, the data acquired by TAP A must be predictively sufficient all the way until it is used by the second invocation of TAP B, as shown by the shaded bar below the OPTION 2 line in Figure D.3.

The increases in efficiency obtained by these two modification options are only possible because predictive sufficiency allows the system to ensure that its functionality is not changed; we cannot arbitrarily cache sensor data, because of the increased sense/act gap. Thus, decisions about the use of internal state result from the principled application of knowledge about the system and the environment in which it is embedded. While information at higher abstraction levels may generally have longer intervals of predictive sufficiency [22], the explicit representation of predictive sufficiency allows the benefits of internal state to be accrued even at lower levels of abstraction.

In sum, predictive sufficiency is a critical concept for embedded agents, because it permits a system to make guarantees about its behaviors. We have shown how CIRCA could use an implementation of predictive sufficiency to guarantee that it will not execute inappropriate actions and that it will react to its environment frequently enough to meet real-time deadlines. A great deal of work remains to be done in implementing this approach and ensuring that all the possible cases of domain interactions are handled correctly.

Explicitly reasoning about predictive sufficiency also allows us to break away from the mindset that decreasing the delay between sensing and acting is always desirable. Specifically, knowing the predictive sufficiency of an observation can allow a system to cache sensory data and maximize the use it gets out of each observation, potentially reducing the frequency of observation and the resulting overhead. We have shown how CIRCA could use this approach to streamline its schedule of reactive behaviors and enhance its real-time performance.

[2] In fact, the second invocation of TAP B is redundant and removable if its tests do not access any other sensors or more-recently-updated memory values.

APPENDIX E

DOMAIN DESCRIPTIONS FOR THE AIS

The following sections list the domain descriptions provided to the AIS for the examples used throughout this thesis. All timing values are listed in microseconds.

The Puma Robot Domain

```
;;----------- Actions for picking part up and putting in box.

(my-make-instance 'action
        :name "pickup_known_part_from_conveyor"
        :preconds '(    (robot_status free)
                        (part_on_conveyor T)
                        (know_type_of_conveyor_part T)
                        (part_in_gripper nil)
                   )
        :postconds '(   (know_type_of_gripper_part T)
                        (part_in_gripper T)
                        (robot_position over_table)
                        (part_on_conveyor nil)
                        (know_type_of_conveyor_part nil)
                    )
        :delay 3500000)

(my-make-instance 'action
        :name "pickup_unknown_part_from_conveyor"
        :preconds '(    (robot_status free)
                        (part_on_conveyor T)
                        (know_type_of_conveyor_part nil)
                        (part_in_gripper nil)
                   )
        :postconds '(   (know_type_of_gripper_part nil)
                        (part_in_gripper T)
                        (robot_position over_table)
                        (part_on_conveyor nil)
                    )
        :delay 3500000)
```

```
;;----------- Moving over box is a process: start, stop, can halt early.

(my-make-instance 'action
        :name "start_moving_over_box"
        :preconds '((robot_status free))
        :postconds '(   (robot_status moving_over_box)
                        (robot_position changing)
                    )
        :delay 20000)

(my-make-instance 'action
        :name "stop_moving_over_box"
        :preconds '(    (robot_status moving_over_box)
                        (robot_position over_box)
                   )
        :postconds '(   (robot_status free)
                        (robot_position over_box)
                    )
        :delay 20000)

(my-make-instance 'action
        :name "stop_moving"
        :preconds '((robot_position changing))
        :postconds '((robot_status free) (robot_position unknown))
        :delay 20000)

;;----------- This TT shows that process of moving over box may eventually
;;             succeed (after at least 2 seconds).

(my-make-instance 'temporal
        :name "arrive_over_box"
        :preconds '((robot_status moving_over_box)
                    (robot_position changing))
        :postconds  '((robot_position over_box))
        :delay 2000000)

(my-make-instance 'action
        :name "place_known_part_in_box"
        :preconds '(    (robot_position over_box)
                        (robot_status free)
                        (know_type_of_gripper_part T)
                        (part_in_gripper T)
                   )
        :postconds '(   (part_in_box T)
                        (part_in_gripper nil)
                        (know_type_of_gripper_part nil)
                    )
```

```
        :delay 4000000)

;;----------- Just putting part on table takes less time:

(my-make-instance 'action
        :name "place_known_part_on_table"
        :preconds '(    (robot_status free)
                        (know_type_of_gripper_part T)
                        (part_in_gripper T)
                   )
        :postconds '(   (part_on_table T)
                        (know_type_of_table_part T)
                        (robot_position over_table)
                        (part_in_gripper nil)
                        (know_type_of_gripper_part nil)
                    )
        :delay 2000000)

(my-make-instance 'action
        :name "place_unknown_part_on_table"
        :preconds '(    (robot_status free)
                        (know_type_of_gripper_part nil)
                        (part_in_gripper T)
                   )
        :postconds '(   (part_on_table T)
                        (know_type_of_table_part nil)
                        (robot_position over_table)
                        (part_in_gripper nil)
                    )
        :delay 2000000)

;;----------- Note the various nondeterministic postconditions: after
;;            picking up a part off table, may be known, unknown, or
;;            no parts left on table.

(my-make-instance 'action
        :name "pickup_known_part_from_table"
        :preconds '(
                        (robot_status free)
                        (part_in_gripper nil)
                        (know_type_of_table_part T)
                        (part_on_table T)
                   )
        :postconds '(   ((part_on_table T)
                         (know_type_of_table_part T)
                         (robot_position over_table)
                         (know_type_of_gripper_part T)
                         (part_in_gripper T)
```

```
                        )
                        ((part_on_table T)
                         (know_type_of_table_part nil)
                         (robot_position over_table)
                         (know_type_of_gripper_part T)
                         (part_in_gripper T)
                        )
                        ((part_on_table nil)
                         (know_type_of_table_part nil)
                         (robot_position over_table)
                         (know_type_of_gripper_part T)
                         (part_in_gripper T)
                        )
                    )
        :delay 3000000)

;;----------- Events for arrival of parts on conveyor.

(my-make-instance 'event
        :name "known_part_arrives"
        :preconds '((conveyor_status free))
        :postconds  '(  (part_on_conveyor T)
                        (conveyor_status busy)
                        (know_type_of_conveyor_part T)
                     ))

(my-make-instance 'event
        :name "unknown_part_arrives"
        :preconds '((conveyor_status free))
        :postconds  '(  (part_on_conveyor T)
                        (conveyor_status busy)
                        (know_type_of_conveyor_part nil)
                     ))

;;----------- After a part has arrived and conveyor goes busy, it can
;;            become free again after some delay (for next 'part slot'
;;            to arrive) and then the event of a part arriving can occur.

(my-make-instance 'temporal
        :name "conveyor_moves_to_next_slot"
        :preconds '((conveyor_status busy))
        :postconds  '((conveyor_status free))
        :delay 50000000)

;;----------- Failure by part falling off conveyor if not processed
;;            before next part arrives.

(my-make-instance 'temporal
```

```
        :name "part_falls_off_conveyor"
        :preconds '((part_on_conveyor T))
        :postconds  '((failure T))
        :delay 50000000)

;;----------- Emergency alert stuff...

(my-make-instance 'event
        :name "emergency_alert"
        :preconds '((emergency nil))
        :postconds  '((emergency T)))

(my-make-instance 'action
        :name "push_emergency_button"
        :preconds '(    (robot_status free)
                        (part_in_gripper nil))
        :postconds '(   (emergency nil)
                        (robot_position over_button))
        :delay 3500000)

(my-make-instance 'temporal
        :name "emergency_failure"
        :preconds '((emergency T))
        :postconds  '((failure T))
        :delay 25000000)

;;----------- Definition of goals.

  (setf *goals* '((part_in_box T)
                    (part_on_conveyor nil)
                    (part_on_table nil)
                    (part_in_gripper nil)
                ))
  (setf *repeat-goals* '( (part_in_box T) ))

;;----------- Definition of initial state.

  (setf *initial-states* (list
        (my-make-instance 'state
                :features '((failure nil)
                            (emergency nil)
                            (know_type_of_conveyor_part nil)
                            (know_type_of_table_part nil)
                            (part_in_gripper nil)
                            (conveyor_status free)
                            (robot_status free)
                            (robot_position over_table)
                            (part_on_table nil)
```

```
                        (part_on_conveyor nil)
                        (part_in_box nil)
            ))
        ))
```

The Bouncing Box Domain

Box 1 is the left box requiring guaranteed, real-time service. Box 2 is the if-time box, which does not lead to failure if it is not serviced by a deadline.

```
;;----------- Actions for bouncing boxes

(my-make-instance 'action
        :name "bounce_box1"
        :preconds '((box1_bounced nil))
        :postconds '((box1_bounced T))
        :delay 10000)

(my-make-instance 'action
        :name "bounce_box2"
        :preconds '((box2_bounced nil))
        :postconds '((box2_bounced T))
        :delay 10000)

;;----------- Boxes should be bounced every few cycles: i.e., they become
;;              unbounced quickly.

(my-make-instance 'temporal
        :name "box1_notbounced"
        :preconds '((box1_bounced T))
        :postconds  '((box1_bounced nil))
        :delay 100)

(my-make-instance 'temporal
        :name "box2_notbounced"
        :preconds '((box2_bounced T))
        :postconds  '((box2_bounced nil))
        :delay 100)

;;----------- It is critical to bounce box1 before some deadline...

(my-make-instance 'temporal
        :name "box1_failure"
        :preconds '((box1_bounced nil))
        :postconds  '((failure T))
        :delay 400000)

;;----------- Action to draw a circle around the mouse-driven cursor.
```

```
(my-make-instance 'action
        :name "mark_cursor"
        :preconds '((cursor_moved_in_window T))
        :postconds '((cursor_moved_in_window nil))
        :delay 12000)


;;----------- Cursor can move as an instant event.

(my-make-instance 'event
        :name "cursor_moves"
        :preconds '((cursor_moved_in_window nil))
        :postconds  '((cursor_moved_in_window T)))


;;----------- If don't track cursor before some time, failure...

(my-make-instance 'temporal
        :name "cursor_failure"
        :preconds '((cursor_moved_in_window T))
        :postconds  '((failure T))
        :delay 900000)

;;----------- Define the task-level goals.

(setf *goals* '((box2_bounced T)(event_not_processed nil)))

;;----------- Define the initial state.

(setf *initial-states* (list
      (my-make-instance 'state
                :features '((failure nil)
                            (cursor_moved_in_window nil)
                            (box1_bounced nil)
                            (box2_bounced nil)
                ))
        ))
```

BIBLIOGRAPHY

[1] P. E. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity," in *Proc. National Conf. on Artificial Intelligence*, pp. 268–272. Morgan Kaufmann, 1987.

[2] P. E. Agre and I. Horswill, "Cultural Support for Improvisation," in *Proc. National Conf. on Artificial Intelligence*, pp. 363–368, July 1992.

[3] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[4] R. C. Arkin, "Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation," in *Robotics and Autonomous Systems 6*, pp. 105–122, 1990.

[5] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–22, March 1986.

[6] R. L. Burden and J. D. Faires, *Numerical Analysis*, PWS-KENT Publishing Co., 1989.

[7] D. Chapman, "Planning for Conjunctive Goals," *Artificial Intelligence*, vol. 32, no. 3, pp. 333–374, July 1987.

[8] J. Connell and P. Viola, "Cooperative Control of a Semi-Autonomous Mobile Robot," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 1118–1121, 1990.

[9] T. Dean and M. Boddy, "An Analysis of Time-Dependent Planning," in *Proc. National Conf. on Artificial Intelligence*, pp. 49–54, 1988.

[10] T. L. Dean, "Intractability and Time-Dependent Planning," in *Proceedings of the 1986 Workshop on Reasoning about Actions & Plans*, pp. 245–266. Morgan Kaufmann Publishers Inc., 1987.

[11] J. DeKleer and J. Brown, "A qualitative physics based on confluences," *Artificial Intelligence*, vol. 24, no. 1-3, pp. 7–83, December 1984.

[12] R. B. Doorenbos, "Matching 100,000 Learned Rules," in *Proc. National Conf. on Artificial Intelligence*, pp. 290–296, 1993.

[13] E. H. Durfee, "A Cooperative Approach to Planning for Real-Time Control," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 277–283, November 1990.

[14] K. Erol, D. Nau, and V. S. Subrahmanian, "When is Planning Decidable?," in *Proc. Int'l Conf. on Artificial Intelligence Planning Systems*, pp. 222–227, 1992.

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

[16] K. Forbus, "Qualitative Process Theory," *Artificial Intelligence*, vol. 24, no. 1-3, pp. 85–168, December 1984.

[17] C. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17–37, September 1982.

[18] M. K. Franklin and A. Gabrielian, "A Transformational Method for Verifying Safety Properties in Real-Time Systems," in *Proc. Real-Time Systems Symposium*, pp. 112–123, 1989.

[19] A. Garvey and V. Lesser, "Design-to-time Real-Time Scheduling," *to appear in* IEEE Trans. Systems, Man, and Cybernetics, vol. 23, no. 6, , 1993.

[20] E. Gat, "ALFA: A Language for Programming Reactive Robotic Control Systems," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, 1991.

[21] E. Gat, *Reliable Goal Directed Reactive Control for Autonomous Mobile Robots*, PhD thesis, Virginia Polytechnic Institute, July 1991.

[22] E. Gat, "On the Role of Stored Internal State in the Control of Autonomous Mobile Robots," *AI Magazine*, vol. 14, no. 1, pp. 64–73, Spring 1993.

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

[24] P. Godefroid and F. Kabanza, "An Efficient Reactive Planner for Synthesizing Reactive Plans," in *Proc. National Conf. on Artificial Intelligence*, pp. 640–645, July 1991.

[25] K. J. Hammond and T. M. Converse, "Stabilizing Environments to Facilitate Planning and Activity: An Engineering Argument," in *Proc. National Conf. on Artificial Intelligence*, pp. 787–793, July 1991.

[26] S. Hanks and R. J. Firby, "Issues and Architectures for Planning and Execution," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 59–70, November 1990.

[27] S. Hanks, "Practical Temporal Projection," in *Proc. National Conf. on Artificial Intelligence*, 1990.

[28] B. Hayes-Roth, "Architectural Foundations for Real-Time Performance in Intelligent Agents," *Journal of Real-Time Systems*, vol. 2, no. 1/2, pp. 99–125, May 1990.

[29] J. Hendler, "Abstraction and Reaction," in *Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[30] 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*, pp. 77–84, November 1990.

[31] 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*, pp. 284–291, November 1990.

[32] L. P. Kaelbling, "Goals as Parallel Program Specifications," in *Proc. National Conf. on Artificial Intelligence*, pp. 60–65, 1988.

[33] L. P. Kaelbling and S. J. Rosenschein, "Action and Planning in Embedded Agents," in *Robotics and Autonomous Systems 6*, pp. 35–48, 1990.

[34] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.

[35] T.-W. Kuo and A. K. Mok, "Load Adjustment in Adaptive Real-Time Systems," in *Proc. Real-Time Systems Symposium*, pp. 160–170, December 1991.

[36] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, "Real-Time Knowledge-Based Systems," *AI Magazine*, vol. 9, no. 1, pp. 27–45, 1988.

[37] J. E. Laird and P. S. Rosenbloom, "Integrating Execution, Planning, and Learning in Soar for External Environments," in *Proc. National Conf. on Artificial Intelligence*, July 1990.

[38] J. E. Laird, "Integrating Planning and Execution in Soar," in *Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[39] J. E. Laird, A. Newell, and P. S. Rosenbloom, "SOAR: An Architecture for General Intelligence," *Artificial Intelligence*, vol. 33, pp. 1–64, 1987.

[40] J. S. Lark, L. D. Erman, S. Forrest, *et al.*, "Concepts, Methods, and Languages for Building Timely Intelligent Systems," *Journal of Real-Time Systems*, vol. 2, no. 1/2, pp. 127–148, May 1990.

[41] 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*, pp. 210–217, December 1987.

[42] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[43] J. W.-S. Liu, K.-J. Lin, and S. Natarajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," in *Proc. Real-Time Systems Symposium*, pp. 252–260, December 1987.

[44] J. W. S. Liu, K.-J. Lin, W.-K. Shih, *et al.*, "Algorithms for Scheduling Imprecise Computations," *IEEE Computer*, vol. 24, no. 5, pp. 58–68, May 1991.

[45] 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*, pp. 198–203, April 1991.

[46] D. M. Lyons, "A Process-Based Approach to Task Plan Representation," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 2142–2147, 1990.

[47] N. Malcolm and W. Zhao, "Version Selection Schemes for Hard Real-Time Communications," in *Proc. Real-Time Systems Symposium*, pp. 12–21, December 1991.

[48] D. McDermott, "A Temporal Logic For Reasoning About Processes and Plans," *Cognitive Science*, vol. 6, pp. 101–155, 1982.

[49] D. McDermott, "Planning Reactive Behavior: A Progress Report," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 450–458, November 1990.

[50] D. McDermott, "A Reactive Plan Language," Technical Report 864, Yale University Department of Computer Science, August 1991.

[51] D. McDermott, "Transformational Planning of Reactive Behavior," Technical Report 941, Yale University Department of Computer Science, December 1992.

[52] D. P. Miller, *Planning by Search Through Simulations*, PhD thesis, Yale University, 1985.

[53] D. P. Miller and E. Gat, "Exploiting Known Topologies to Navigate with Low-Computation Sensing," in *Proc. SPIE Sensor Fusion Conf.*, November 1990.

[54] D. J. Musliner, E. H. Durfee, and K. G. Shin, "Any-Dimension Algorithms," in *Proc. Workshop on Real-Time Operating Systems and Software*, pp. 78–81, May 1992.

[55] D. J. Musliner, E. H. Durfee, and K. G. Shin, "Any-Dimension Algorithms and Real-Time AI," Technical Report CSE-TR-151-92, University of Michigan Computer Science and Engineering, December 1992.

[56] 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*, pp. 104–107, March 1992.

[57] D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: A Cooperative Intelligent Real-Time Control Architecture," *to appear in* IEEE Trans. Systems, Man, and Cybernetics, vol. 23, no. 6, , 1993.

[58] P. Nii, "The Blackboard Model of Problem Solving," *AI Magazine*, vol. VII, no. 2, pp. 38–53, Summer 1986.

[59] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Press, Palo Alto, CA., 1980.

[60] C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider, "Reducing Problem-Solving Variance to Improve Predictability," *Communications of the ACM*, vol. 34, no. 8, pp. 81–93, August 1991.

[61] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, 1981.

[62] J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[63] S. J. Rosenschein, "Synthesizing Information-Tracking Automata from Environment Descriptions," Technical Report 2, Teleos Research, July 1989.

[64] 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*, pp. 83–98, 1986.

[65] S. J. Russell and S. Zilberstein, "Composing Real-Time Systems," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 212–217, August 1991.

[66] M. J. Schoppers, "Universal Plans for Reactive Robots in Unpredictable Environments," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 1039–1046, 1987.

[67] M. Schoppers, "Automatic Synthesis of Perception Driven Discrete Event Control Laws," in *Proc. 5th IEEE Int'l Symposium on Intelligent Control*, pp. 410–416, September 1990.

[68] M. Schoppers, "Introduction to Special Edition on Real-Time Knowledge-Based Control Systems," *Communications of the ACM*, vol. 34, no. 8, pp. 27–30, August 1991.

[69] 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*, July 1992.

[70] R. Simmons, "An Architecture for Coordinating Planning, Sensing, and Action," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 292–297, November 1990.

[71] R. Simmons, "Robust Behavior with Limited Resources," in *Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[72] R. Simmons, "Coordinating Planning, Perception, and Action for Mobile Robots," in *AAAI Spring Symposium*, 1991.

[73] H. A. Simon, *Models of Bounded Rationality*, M. I. T. Press, 1982.

[74] *pASSPORT+: A pSOS+-based Environment for Real-Time Software Development*, Software Components Group, Inc., 1990.

[75] M. H. Soldo, "Reactive and Preplanned Control in a Mobile Robot," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 1128–1132, 1990.

[76] J. A. Stankovic, "Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, vol. 21, no. 10, pp. 10–19, October 1988.

[77] S. Vere, "Temporal Scope of Assertions and Window Cutoff," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 1055–1059, 1985.

[78] E. Walden and C. V. Ravishankar, "Algorithms for Real-Time Scheduling Problems," Technical Report CSE–TR–92–91, University of Michigan, Computer Science and Engineering, April 1991.