CIRCA:
A Cooperative Intelligent Real-Time Control Architecture

David J. Musliner Edmund H. Durfee Kang G. Shin

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

{djm, durfee, kgshin}@eecs.umich.edu
(313) 763-5363

ABSTRACT

Most research into applying AI techniques to real-time control problems has limited the power of AI methods or embedded "reactivity" in an AI system. We present an alternative, cooperative architecture that uses separate AI and real-time subsystems to address the problems for which each is designed; a structured interface allows the subsystems to communicate without compromising their respective performance goals. By reasoning about its own *bounded reactivity*, CIRCA can *guarantee* that it will meet hard deadlines while still using unpredictable AI methods. With its abilities to guarantee or trade off the timeliness, precision, confidence, and completeness of its output, CIRCA provides more flexible performance than previous systems.

Index Terms: Real-Time Control; Artificial Intelligence; Reactive Systems; Resource Scheduling; Planning; Cooperation; Intelligent Robotics.

To appear in
IEEE Transactions on Systems, Man, and Cybernetics, vol. 23, no. 6, 1993.

The work reported in this paper was supported in part by the National Science Foundation under Grants DMC-8721492 and IRI-9158473, by the Office of Naval Research under Grant N00014-91-J-1115, and by a NSF Graduate Fellowship. The opinions, findings, and recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the funding agencies.

1 Introduction

As Artificial Intelligence (AI) techniques become mature, there has been growing interest in applying these techniques to controlling complex real-world systems which involve hard deadlines. In such systems, the controller is required to respond to certain inputs within rigid deadlines, or the system may fail catastrophically. Since the number of possible domain situations is too large to be fully enumerated, and the consequences of failure are so severe, testing alone is insufficient to guarantee the required real-time performance [24, 40]. These control problems require systems which can be *proven* to meet the hard deadlines imposed by the environment. Unfortunately, many AI techniques and heuristics are not suited to analyses that would provide guaranteed response times [11]. Even when AI techniques can be shown to have predictable response times, the variance in these response times is typically so large that providing timeliness guarantees based on the worst-case performance would result in severe underutilization of the computational resources during normal operations [33].

Thus we perceive an apparent conflict between the nature of AI and the needs of real-world, real-time control systems. While AI methods are characterized by unpredictable or high-variance performance, real-time control systems require constant, predictable performance. Most research on "real-time AI" focuses either on restricted AI techniques that have predictable performance characteristics [4, 19, 23] or on reactive systems that retain little of the power of traditional AI [1, 5]. Several researchers are investigating systems which combine reactive and traditional AI methods [2, 14, 31, 35]. These approaches have concentrated on retaining both reactive and unpredictable mechanisms, but do not address the guarantees required by hard real-time tasks.

To combine unrestricted AI techniques with the ability to make hard performance guarantees, we propose a Cooperative Intelligent Real-time Control Architecture (CIRCA). In this architecture, an AI subsystem reasons about task-level problems that require its powerful but unpredictable reasoning methods, while a separate real-time subsystem uses its predictable performance characteristics to deal with control-level problems that require guaranteed response times. The key difficulty with this approach is allowing the subsystems to interact without compromising their respective performance goals. We have developed a scheduling module and a structured interface that allow the unconstrained AI subsystem to asynchronously direct the real-time subsystem without violating any response-time guarantees.

Realistic intelligent control systems must recognize their resource limitations and make tradeoffs in the quality of their control outputs, or responses. Section 2 of this paper briefly discusses the previous approaches that have reasoned about such limitations, and describes CIRCA's unique ability to guarantee a chosen subset of responses. CIRCA's scheduling module allows the AI subsystem to reason explicitly about the real-time subsystem's execution resources, and the response guarantees it can provide. Since these guarantees are based on worst-case performance measures, CIRCA also provides mechanisms to utilize the time which becomes available when guaranteed mechanisms use less than their scheduled time allowance.

Section 3 develops a functional distinction between task-level goals and control-level goals, and presents a formal graph model of CIRCA's interactions with its environment. This model forms the basis for CIRCA's performance guarantees, and illustrates how CIRCA makes performance tradeoffs based on resource limitations.

Section 4 presents the architecture itself, and describes the interface which allows the AI subsystem and the real-time subsystem to cooperatively guarantee deadlines and achieve goals. This section also presents a formal statement of the guarantees that CIRCA can provide. Section 5 describes our prototype implementation, which incorporates an AI subsystem combining features of both the PRS [13, 20] and blackboard [32] architectures. The prototype system pilots a Heathkit Hero 2000 robot through the hallways of our building. The robot uses sonar to sense its orientation with respect to the hallway, as well as to sense obstacles and the open/closed status of doorways along its path. Throughout the paper, we use examples from this domain to illustrate both the concepts underlying the architecture and the details of our prototype implementation. Section 6 presents a more detailed comparison to related work, and Section 7 concludes with a brief discussion of future research directions.

2 Performance Tradeoffs and Bounded Reactivity

The responses of an intelligent control system can be rated along four dimensions: completeness, precision, confidence, and timeliness [28]. Completeness means that responses are produced for all possible inputs; timeliness means that the responses are produced before any associated deadlines. Precision and confidence together determine the "quality" of a solution, or how accurate the output is, to the best of the system's knowledge. An ideal intelligent control system could guarantee that any possible sequence of inputs would elicit optimal responses from the system, within all timing requirements.

Some systems strive for this ideal by assuming they have unlimited processing resources. For example, the subsumption architecture [5] assigns each reactive element to a separate processor. Such assumptions limit scalability: it would be highly impractical to build a subsumption system to control an oil drilling platform, which can make up to 20,000 signals available to its operators [3, 24].

Other systems recognize that processor limitations make realistic control systems subject to the same "bounded rationality" [38] as humans, pushing ideal performance out of reach. To deal with bounded rationality, these systems provide differing levels of guarantees for the four performance dimensions. The guarantees that a system provides are often defined by the conditions that determine when its control algorithm returns a result. We call methods that halt when they reach a certain threshold along a dimension "any-<dimension>" algorithms. For example, "any-time" algorithms can be terminated at any time, yielding some result, possibly with reduced precision, confidence, or completeness [4, 7, 19]. If "any-time" algorithms are interrupted before the deadline for every response, they guarantee timeliness and completeness. Many iterative numerical methods [6] are "any-precision" algorithms that terminate when a result with a certain precision has been achieved. Similarly, algorithms that halt when the confidence in a solution rises above a threshold are examples of "any-confidence" algorithms.

These types of systems are inappropriate for hard real-time control tasks because they cannot guarantee *acceptable* results within a deadline. Even "any-time" systems are inappropriate, because they have no control over the degree of response quality degradation which may occur. A hard real-time control system might only guarantee a subset of tasks, but that subset requires guaranteed timeliness, precision, and confidence to ensure that the system does not fail catastrophically. Realistic systems must also recognize that, in addition to processor limitations, sensor and actuator limitations constrain intelligent control systems. Even if a system's processors are fast enough, its sensors and actuators might not be able to provide ideal performance. Thus a system must recognize its "*bounded reactivity*" as well as its bounded rationality.

CIRCA was designed to meet these demands 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 in order to achieve precision, confidence, and timeliness. CIRCA reasons about its bounded reactivity within the AI subsystem (AIS) and the Scheduler, which cooperate to decide which responses the real-time subsystem (RTS) can and should guarantee. The AIS and the Scheduler form an "any-completeness" system, searching for a subset of guaranteed responses that will cover the inputs which are expected to occur in the domain at each moment. If the AIS can move the subset of guaranteed responses over the complete response set properly, CIRCA will provide guaranteed ideal performance.

3 Making Performance Guarantees

How can CIRCA provide any performance guarantees when its AIS uses high-variance or unpredictable computations? The answer is based on the distinction we draw between task-level goals and control-level goals. This distinction is largely a functional one: in CIRCA, the RTS uses predictable methods to achieve control-level goals, while the AIS has unpredictable AI techniques to decompose task-level goals into control-level goals. *CIRCA is designed to reason about guaranteeing its control-level goals, but not necessarily its task-level goals.*

The choice of which specific goals are assigned to which category is largely up to the system designer: the "control-level" and the "task-level" are somewhat arbitrary divisions along a continuous range of problem complexities. However, since CIRCA's guarantees are based on worst-case performance assumptions, assigning goals which need high-variance algorithms to the RTS results in a decreased capacity to guarantee other control-level goals. Thus the system designer must decide which types of goals will require guarantees, and which can be left less predictable. Given that separation, the AIS and Scheduler attempt to guarantee the control-level goals.

We have developed a graph model that represents the AIS' reasoning about the RTS and its environment. The graph model shows how we can recognize what subset of reactions are necessary to satisfy the system's control-level goals. Thus the model allows us to concisely state the assumptions that CIRCA requires to guarantee its control-level goals (see Section 4.6). Furthermore, the model shows how a subset of all reactions can isolate the guaranteed control level from the unguaranteed task level, so that the unpredictable AIS does not cause the RTS to violate hard deadlines.

3.1 The Graph Model of RTS/Environment Interaction

The directed graph model represents the *worst-case* behavior of the environment, and the actions which the RTS can take to avoid failure. The graph model has five elements (S, F, T_E, T_A, T_T):

1. A finite set of "states" $S = \{S_1, S_2, ..., S_m\}$, where each state S_i represents a description of relevant features of the world.

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

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

An "event-closed" set of states $S_{EC} \subseteq S$ is defined as a connected 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, non-temporal events (as opposed to the mere progression of time) cannot move the system out of the event-closed set of states.

An event-closed set of states which does not contain the failure state is called a "safe" set of states ($F \notin S_{safe}$). Note that a safe set of states can still lead to failure 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 enters a new state because of an event, but fails to react to the new state before a hard deadline, it will have entered the failure state via a temporal transition: that is, by "waiting too long" to react, the system fails.

Figure 1 shows a portion of the graph model for the example hallway-following task. Each labeled box in the graph model represents a state S_i. Solid single arrows represent event transitions T_{Ei}, dashed single arrows represent action transitions T_{Ai}, and double arrows represent temporal transitions T_{Ti}. In the figure, states {A,B,C,D,E} form a safe set of states S_{safe}: no event transitions lead out of the set. However, temporal transitions can still lead to failure. In the example, we can see this in the transition from state B to state C caused when an obstacle appears in the path; if the system waits too long to recognize the situation and take action, it will follow the temporal transition to F by colliding with the obstacle. But, if the system quickly detects the obstacle and halts, it can avoid a collision and transition to state E instead. Thus, if the system can guarantee that it will always preempt temporal transitions that lead from states in the safe set to the failure state, moving instead to another state within the safe set, then *the system can remain in a safe set of states for an indefinite period of time without violating its control-level goals or the domain constraints.* The big "if," then, requires that the system provide the appropriate action transitions to stay within the safe set. CIRCA was designed to achieve just that, using the AIS and Scheduler to reason about which transitions to guarantee, and the RTS to implement those guaranteed transitions.

In our prototype implementation, described in Section 5, the AIS reasons *implicitly* about the information in the graph model to derive the necessary RTS reactions. We are investigating a generalized production representation for the graph transitions that may allow the AIS to reason

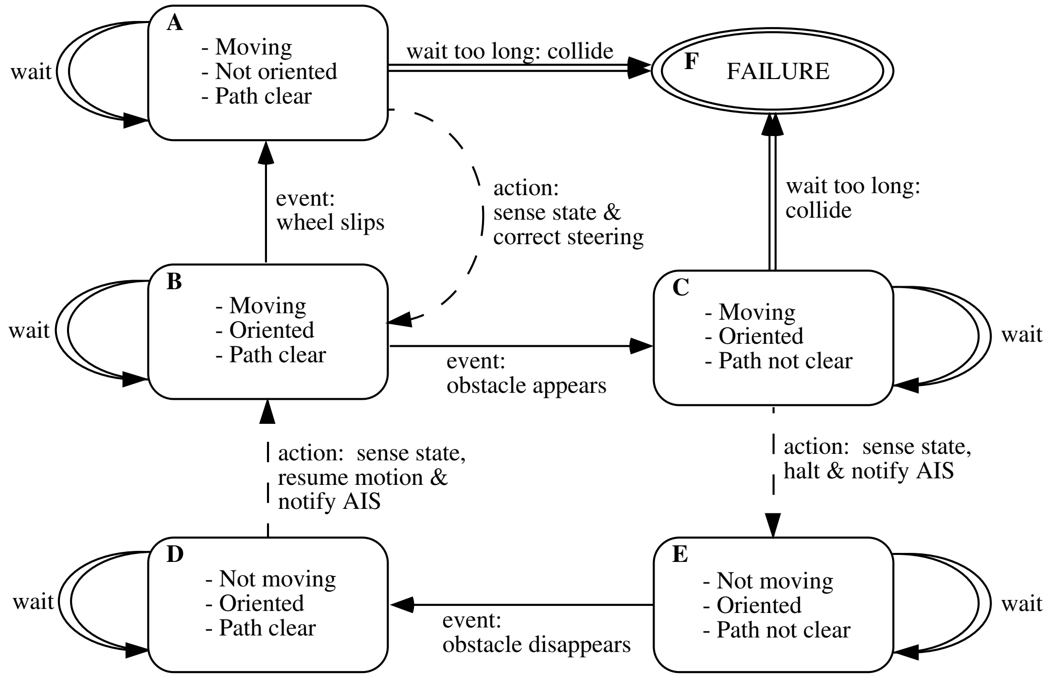

Figure 1: An example portion of the graph model of RTS/environment interactions. Some states and transitions are omitted for clarity.

explicitly about the graph model without encountering the state space explosion associated with enumerating all possible world states.

3.2 Model Requirements

The example graph model shows only deterministic transitions, where a given state and input (action, event, or time) always determine a unique new state. However, nothing prevents us from modeling non-deterministic transitions, which can map a given state and input into one of several new states. We are not concerned with actually simulating the behavior of the world, but rather with extracting its worst-case behavior and the minimum time to failure from various world states. Thus non-determinism only provides more paths to consider when deriving worst-case behaviors.

To allow guaranteed preemption of temporal transitions to failure, we must place two requirements on the behavior of the world. First, we must know the minimum duration of any world states that require responses, so that we can derive a sampling frequency sufficiently high to detect the state. Second, if failure may result from performing an action required by a state after that state has been left, we must ensure that our sensors have a predictive capability that spans the gap between sensing a state and acting on that information. 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. Both of these assumptions are required by any system striving to make similar guarantees.

3.3 Accounting for Incomplete Knowledge and Bounded Rationality

Since the graph model represents the AIS' reasoning about the RTS and the environment, it is inherently subject to the AIS' incomplete domain knowledge. The model only represents those states and transitions which the AIS can reason about. For example, the prototype AIS has no knowledge to indicate that the Hero's battery may become discharged, and thus the AIS does not reason about that possibility, its graph model would not include that transition to F, and it does not arrange responses to avoid that type of failure. This is perfectly acceptable, since no system can plan to avoid an event for which it has no relevant knowledge.

The AIS might recognize its own bounded rationality, and further limit the extent of the graph model it considers, in order to speed processing. For example, the AIS could incorporate an "any-time" algorithm which would incrementally expand the graph model it used to derive the guaranteed set of responses for the RTS. The AIS would start with only the most probable states and transitions, and would add new, less probable states to the model only after a guaranteed response set was generated for the previous partial model. If the AIS ran out of time before examining the entire set of states that the graph model could potentially involve, it would at least have built a set of responses to cover a highly-probable subset of the safe set.

3.4 Accounting for Bounded Reactivity

As discussed in Section 2, a realistic control system also has bounded reactivity. The AIS can account for the limitations on sensor and actuator resources available to the RTS by effectively modifying its graph model. For example, if the AIS attempts to guarantee responses to preempt both of the temporal transitions to failure shown in Figure 1, the Scheduler may indicate that the RTS does not have sufficient sensor resources to guarantee both responses (it cannot sense for both orientation and obstruction frequently enough). The AIS can then modify its model to decrease the sensor requirements, possibly by eliminating low probability transitions associated with the oversubscribed sensor. In this example, the AIS might stop considering the transition from B to C, eliminating the demands for sensors to check that transition. This solution trades off completeness against guaranteed timeliness, so that the system can no longer guarantee its safety from all known forms of failure.

As an alternative approach to modifying the model to account for resource limitations, the AIS might change a parameter which affects the deadlines associated with temporal transitions to failure. For example, if the robot's speed is decreased, the time before the transition from state C to F will be increased. Thus the RTS would not have to check for obstacles ahead as frequently, and the demands on the sensors would decrease.

This process of reasoning about resource constraints is CIRCA's mechanism for making tradeoffs in the completeness of its guaranteed responses. The system strives to guarantee to preempt all known transitions to failure; when it cannot, it modifies the modeled set of states and transitions, and thus alters the required subset of guaranteed responses.

4 CIRCA

Figure 2 illustrates the Cooperative Intelligent Real-Time Control Architecture. The RTS is responsible for implementing the actual guaranteed responses; the AIS and the Scheduler cooperate

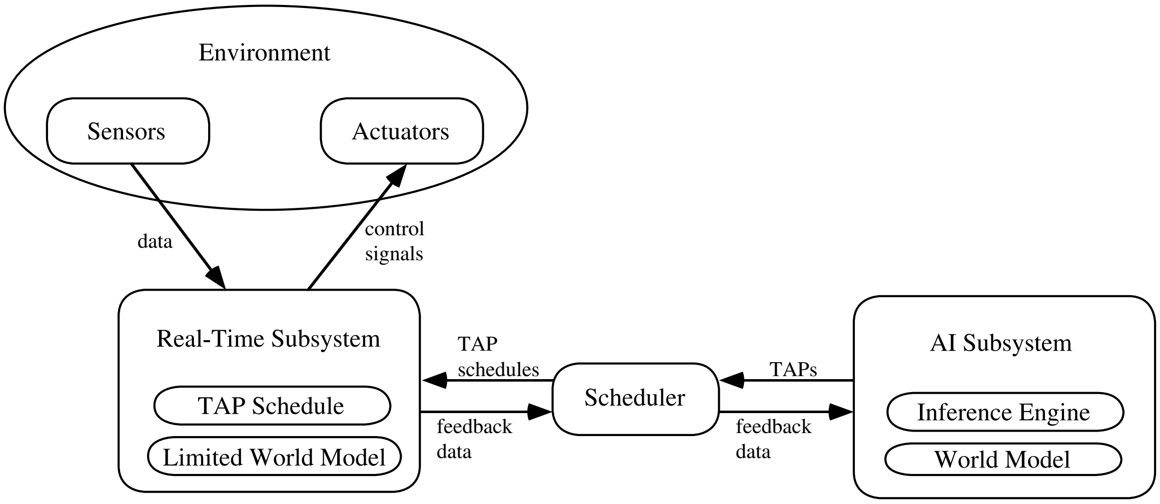

Figure 2: The Cooperative Intelligent Real-Time Control Architecture.

to adjust the subset of responses that the RTS is supporting, attempting to ensure that the overall system meets hard deadlines and also achieves system goals as closely as possible.

The RTS executes a cyclic schedule of simple test-action pairs (TAPs[1]) which have known worst-case execution times. Since the RTS performs no other functions, it can guarantee that the scheduled tests and actions are performed within predictable time bounds. The AIS reasons about the RTS' bounded reactivity, attempting to find a subset of TAPs which can be guaranteed to meet the control-level goals and make progress towards the task-level goals. In cooperation with the AIS, the Scheduler reasons about the limited execution resources available to the RTS, and builds the schedule of TAPs. Since the AIS and RTS run asynchronously, the AIS need not conform to the rigid performance restrictions which the RTS uses to guarantee meeting hard deadlines. Thus the AIS can utilize unpredictable, high variance heuristics without compromising the overall system's ability to meet real-time deadlines.

4.1 Test-Action Pairs (TAPs)

The TAP structure was developed to provide a standard primitive with which the Scheduler and AIS can reason about the timing and resource characteristics of the RTS' behavior, in order to guarantee meeting deadlines and resource restrictions. An implementation of CIRCA defines a number of TAP classes, and the RTS runs schedules made up of instances of these classes. Each TAP class has a fixed set of tests (or preconditions), a set of actions to take if all the tests return true, data about the sensing and actuating resources the TAP requires, and worst-case timing data on how long it takes to test the preconditions and execute the actions. Specific TAP instances also have parameters which the AIS can set depending on the context in which the TAP is used. These parameters include frequency requirements or deadlines, which constrain the times at which a TAP must be invoked. TAP tests and actions may include acquiring sensor readings, performing limited data processing, and controlling the system actuators.

For example, Figure 3 shows an instance of the simple **stop-if-object-ahead** TAP class used in the hallway-following task. The TEST specifies that the TAP should only be executed if the robot

[1]Not to be confused with Firby's RAPs [12], which are larger in scale and do not have predictable execution times.

```
TAP stop-if-object-ahead
    :TEST (and *moving* (< (get-sonar-reading-ahead) *safety-distance*))
    :ACTION (progn (halt) (notify-AIS 'halted))
    :RESOURCES (sonar base-motors)
    :MAX-PERIOD .70     :TEST-TIME .15     :ACTION-TIME .05
```

Figure 3: A **stop-if-object-ahead** TAP instance.

is moving and the distance ahead of the robot, as sensed by sonar, is less than `*safety-distance*`. If these conditions are true, the robot is halted and the RTS sends a notification message to the AIS. Testing and executing this TAP takes a maximum of .20 seconds (TEST-TIME + ACTION-TIME), and the AIS has determined that it must be run at least every .70 seconds (MAX-PERIOD) to guarantee avoiding collisions with objects in front of the robot. Note that the frequency with which this TAP must be executed is dependent on the speed of the robot's motion: the AIS reasons about this and other variables to produce the parameters such as MAX-PERIOD, which can be different in each TAP instance.

4.2 The AI Subsystem (AIS)

The AIS decomposes task-level goals into plans consisting of several phases. The AIS reasons implicitly about the safe set of states which the RTS might encounter during each plan phase, and tries to generate a set of TAPs which will preempt the temporal transitions to failure, so that the RTS remains in the safe set. In our model of the AIS/Scheduler cooperation, the AIS suggests this set of TAPs to the Scheduler, which attempts to build a TAP schedule. The Scheduler returns either a successful schedule that meets all the timing constraints, or some informative feedback if it fails to produce such a schedule. In that case, the AIS will modify the suggested set of TAPs, possibly by altering timing parameters, by choosing alternate TAPs to produce a desired behavior, or by actually dropping some TAPs altogether. In this way, the AIS and Scheduler reason about the RTS' bounded reactivity, and choose how to degrade performance to meet those limitations.

The AIS and the Scheduler run in parallel with the RTS, trying to form the TAP schedule for the next plan phase before the RTS has accomplished the goals of the current phase. The AIS can immediately download the new TAP schedule to the RTS, and the RTS will switch to the new schedule when the current goals are achieved (see Section 4.4). If the AIS and Scheduler do not produce the new TAP schedule before the RTS has accomplished the goals of the current schedule (possibly because of unexpected events), the RTS continues to execute the current schedule, waiting in a safe set of states. Thus the RTS' guarantees of system safety are decoupled from the AIS' unpredictable performance.

4.3 The Scheduler

The Scheduler is responsible for building a cyclic schedule from the list of TAPs suggested by the AIS. The Scheduler reasons about the maximum periods of the TAPs, their worst-case execution time and resource needs, and the resources available from the RTS. If the Scheduler cannot build a successful schedule to guarantee all the TAP timing constraints, it returns a failure message to the AIS, which must adjust its proposed TAP list to loosen the scheduling constraints.

```
(load-bootstrap-schedule)
(setf *run-current-schedule* T)
(while T (while *run-current-schedule* (process-TAP-schedule *schedule*))
         (setf *schedule* *new-schedule*)
         (setf *next-schedule-available* nil)
         (setf *run-current-schedule* T))
```

Figure 4: The RTS main loop.

```
TAP check-for-new-schedule
    :TEST (msg-pending-from-AIS)
    :ACTION (progn (read-bytes-from-AIS *packet-size* *new-schedule*)
                   (if (schedule-complete *new-schedule*)
                       (setf *next-schedule-available* T)))
    :MAX-PERIOD 1.5     :TEST-TIME .10     :ACTION-TIME .15
```

Figure 5: A **check-for-new-schedule** TAP instance.

The Scheduler builds schedules based on the assumption that each TAP will use its worst-case time to test and execute. But, when the RTS is running, many TAPs test negative and are not executed. Their scheduled execution time is available for other RTS activity. To fill in these unpredictable blocks of free time, the AIS sends a special "unguaranteed" list of TAPs to the RTS along with the guaranteed TAP schedule. When TAPs don't fire, the RTS can look for unguaranteed TAPs that fit the remaining scheduled execution time[2], and invoke them instead. These unguaranteed TAPs provide a form of graceful degradation: as the system becomes more time-pressured, they will be run less frequently.

4.4 The Real-Time System (RTS)

The RTS is actually a fairly simple program loop, shown in Figure 4, that loads and then executes the TAP schedule sent from the AIS. Since the RTS has no other duties, its performance exactly follows the model which the Scheduler used to produce the TAP schedule. The RTS runs through the guaranteed TAP schedule, checking the tests for each TAP and firing those TAPs whose tests return true. If a TAP is not fired, the RTS uses the time scheduled for its execution to search for and invoke one or more of the unguaranteed TAPs.

But, if the RTS is only executing the TAP schedule, how does it ever get a new TAP schedule? The answer is simple: the TAP schedule itself includes the **check-for-new-schedule** TAP, shown in Figure 5, which causes the RTS to check for a new TAP schedule from the AIS. In other words, the AIS actually determines how often the RTS is checking to see if a new schedule is available. Thus, 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 **check-for-new-schedule** TAP.

The **check-for-new-schedule** TAP is also important because it reads in the new TAP schedule

[2] Actually, the unused time minus the time needed to find an appropriate unguaranteed TAP.

```
TAP end-hallway
    :TEST (>= (get-distance-traveled) *traversal-distance*)
    :ACTION (cond (*next-schedule-available*
                    (setf *run-current-schedule* nil))
                  (*moving* (halt) (notify-AIS 'reached-end)))
    :MAX-PERIOD 1.5     :TEST-TIME .15     :ACTION-TIME .10
```

Figure 6: An **end-hallway** TAP instance.

incrementally. Each time the TAP's test indicates there is data waiting from the AIS, the RTS will read in a constant amount 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.

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. In other words, 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. More likely, 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 schedule is available. Figure 6 shows an example TAP that detects when the robot has traveled the intended distance. If the schedule for the next plan phase is available, the TAP sets a flag causing the RTS to terminate the current schedule, dropping back into the main RTS loop shown in Figure 4. At that point, the RTS makes three variable assignments and immediately returns to executing the TAP schedule. Thus, the transition between TAP schedules is extremely rapid, and can be subsumed by the execution time requirements of the **end-hallway** TAP without great cost.

The output channel from the RTS to the AIS is also used only within TAPs, as illustrated in Figure 6. The **end-hallway** TAP includes an explicit message-sending function, 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 in and out of the RTS is scheduled explicitly within TAPs, avoiding unpredictable I/O delays. This not only allows the Scheduler to make guarantees, 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 [17].

In general, the RTS might run on multiple processors, as long as the Scheduler could model its performance correctly. We have presented the RTS and TAP interface for a single processor; the principles of predictable primitives and incremental, scheduled communication would extend in a straightforward manner. However, if more than one TAP could be active simultaneously, issues of command fusion and resource access arbitration would complicate the predictability. Reactive architectures which assume numerous parallel behaviors have been developed to address these problems [1, 5, 34].

4.5 How CIRCA Meets the Requirements for Real-Time Intelligent Systems

Based on an extensive survey of current research and development, Laffey *et al.* [24] outlined several requirements for real-time intelligent systems. In this section, we show how CIRCA meets most of these requirements explicitly, and provides a framework within which advanced AI techniques can address the remaining goals.

Integration of Numeric and Symbolic Computing: Highly predictable, numeric algorithms for signal processing and feature extraction can be easily invoked by TAPs, and their results can be utilized by the reactive RTS as well as transmitted to the symbolically-oriented AIS. CIRCA provides the TAP formalism to express the control of such algorithms, and the Scheduler reasons about the integration of their performance with sensing and acting behaviors.

Interrupt Handling: Many systems use hardware interrupts to asynchronously signal important events. As discussed above, the RTS is expected to have TAPs that check for all important events as frequently as necessary. Thus, the RTS does not need hardware interrupts; in fact, allowing interrupts would make the RTS' performance unpredictable, voiding CIRCA's guarantees. In a sense, we have moved the loop which polls for important events 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 polling loop. Moving the polling loop decreases its frequency, since many processor instructions are involved in running each TAP, but increases the architecture's ability to control and predict the loop.

Resource Utilization: The Scheduler reasons explicitly about the computing, sensing and actuating resources available to the RTS, so it can schedule optimal resource utilization. The ongoing cooperation between the AIS and the Scheduler allows CIRCA to trade off any of its performance dimensions for any other, based on the system goals and the resource limitations.

Temporal Reasoning and Truth Maintenance: The prototype AIS, described in Section 5.1, currently includes rudimentary mechanisms to reason about relations between time points. The AIS uses these mechanisms to reason about the temporal relations between tasks, so that it can focus its attention on the earliest-deadline task during time-pressured situations. Nothing prohibits the integration of truth maintenance systems [10] or more efficient temporal reasoning systems [9, 22] into the AIS.

Continuous Operation and Fault Tolerance: The architecture uses asynchrony, the TAP interface, and safe sets of states to isolate the predictable RTS (which is continuously reactive) from the unpredictable AIS (which is continuously deliberative). CIRCA can detect faults in its sensors, actuators or the RTS by having the Scheduler monitor those subsystems (without interfering with the RTS), or by having explicit monitoring TAPs built into the TAP schedule. The AIS and Scheduler can then modify their models of those resources, taking faults into account and automatically adjusting their performance. In situations where the AIS and RTS are physically separated, the Scheduler's monitoring behaviors might be best implemented on the same platform as the RTS. CIRCA supports this possibility by assuming only a loose coupling between the cooperating Scheduler and AIS.

Predictability: CIRCA was primarily designed to achieve the goal of predictability. By reasoning explicitly about the detailed, predictable behaviors described by TAPs, CIRCA can provide guaranteed timeliness, precision, confidence, and completeness, if sufficient resources are available. And, when resource restrictions prohibit ideal performance, CIRCA can explicitly reason about trading off these performance dimensions to achieve guarantees for a subset of control-level goals.

4.6 The Control-Level Guarantee

As discussed in Section 3, CIRCA strives to guarantee its control-level goals. Of course, all guarantees must make some assumptions, and the value of a guarantee depends on how well understood and reasonable those assumptions are. We now describe the assumptions which underlie the architecture's control-level guarantees, and argue that they are quite reasonable.

Assuming that

$\mathcal{A}1$) the space of system states can be covered by safe sets of states, and

$\mathcal{A}2$) the RTS is given complete TAP schedules to preempt the temporal transitions to failure in each safe set, then

$\mathcal{G}1$) CIRCA can guarantee that it will never violate a control-level goal or domain constraint.

Assumption $\mathcal{A}1$ is really not very restrictive, since the only type of transition that cannot be covered by a safe set of states is a non-temporal event transition to failure. This is reasonable, because if there is no time delay between an event and failure, the system cannot possibly preempt the failure. For example, if our model of the environment includes the possibility of a nuclear blast very near the robot, no control system could guarantee avoiding failure. Essentially, requiring coverage by safe sets of states merely insists that it is possible for the agent to guarantee survival in its environment.

Assumption $\mathcal{A}2$ requires that the AIS and Scheduler manage to produce a successful, complete TAP schedule for each phase of a plan before the RTS begins executing that phase. By the definition of a safe set of states, no non-temporal events can move the system out of the safe set. And, since the previous TAP schedule is guaranteed to preempt all temporal transitions out of the safe set, the RTS can remain in a safe set of states for an indeterminate period without failure. Therefore, the AIS is not *required* to produce TAP schedules before the RTS has accomplished the purpose of a previous schedule. For example, once the Hero has traversed a hallway it may wait (safely) at the end of the hall for a new TAP schedule. Thus assumption $\mathcal{A}2$ is reduced to a simple requirement that the AIS and Scheduler produce the TAP schedule at *some* time.

Furthermore, if we assume that

$\mathcal{A}3$) the AIS and Scheduler perform a complete search of the space of TAP schedules,

then $\mathcal{A}2$ reduces to a requirement that some complete TAP schedule exists for each safe set of states. This is equivalent to saying that the RTS has sufficient resources to guarantee control-level goals in the given environment, and of course that assumption must underlie all systems which make such guarantees. We can thus restate $\mathcal{G}1$ as follows: if the system has the requisite resources and it is possible to guarantee survival in the environment, then CIRCA can guarantee all its control-level goals.

4.7 The Task-Level Guarantee?

The control-level guarantee is based largely on the fact that the system can remain in a safe set of states for an indeterminate amount of time without violating its control-level goals. Of course, the same cannot be said for task-level goals: the AIS must produce the necessary TAP schedules fast enough that the system achieves its task-level goals within their deadlines. Guaranteeing task-level goals presents several difficulties, not the least of which is defining "fast enough" in the previous sentence.

One problem with reasoning about the task level results from task decomposition: when a high-level goal is broken up into sequences of subgoals, and finally into TAPs, the time available to achieve the high-level goal must also be split into intervals for each subgoal. Doing this correctly would require the AIS to have a predictable model of how long it takes to solve a subgoal. But we have already noted that AI methods are either too unpredictable or too high-variance to be usefully modeled. Thus the AIS cannot use unpredictable methods and also reason accurately about apportioning time to subgoals.

Even if we restricted the complexity of the AIS to predictable methods, the time required to actually achieve each subgoal in the real world is dependent on the dynamic environment. Thus, unexpected events might delay the accomplishment of a subgoal, violating the subgoal's time interval. In many cases, this violation need not be a fatal event; the AIS might be able to manipulate the time spent on later subgoals so that the overall high-level goal is still achieved before its deadline. For example, a delivery task with a hard deadline might be broken into several hallway traversal tasks. If the Hero's path is temporarily blocked during one of those traversals, the system might need to adjust the subsequent traversals to a higher speed, to make up for the lost time. This adjustment would probably reduce the safety of the system, sacrificing safety for timeliness. This type of dynamic performance tradeoff requires complex task-level reasoning that varies with the environment, and thus defies *a priori* analyses and guarantees.

5 The Prototype Implementation

We have built a prototype CIRCA system which emphasizes the interface between the subsystems, bridging the gap between the uncertain performance of the AIS and the rigid constraints of the RTS. The prototype is designed to provide control-level guarantees based on the graph model which underlies the architecture's design. And, 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.

5.1 The Prototype AI Subsystem

The design of the prototype AIS is motivated by several operational requirements. To decide what subset of all possible TAPs should be active at any given time, the AIS must reason about the system goals, the current world state, the anticipated future world states, and the capabilities of the RTS. Our implementation reasons about a declarative world model built upon classes of "objects" and "tasks." Object classes and instances are used to represent the physical environment and the system's sensor and actuator resources. Task classes and instances provide a hierarchic representation of the goals which the system is pursuing, their temporal relations, and the progress

```
(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, climb to meta-level.
               (setf *soak* *new-soak*))))
```

Figure 7: The prototype AIS interpreter.

made so far.

Since CIRCA is designed to operate in highly dynamic environments where plans may fail and unexpected circumstances may arise at any time, the prototype AIS should be highly interruptible, and allow rapid context switches to reasoning about new, unexpected problems. The prototype AIS must be able to reason about multiple goals and about multiple methods for achieving those goals, so that it can evaluate tradeoffs of the various performance dimensions based on resource limitations, event probabilities, criticality measures, etc.

The prototype inferencing mechanism incorporates features derived from the PRS [13, 20] and blackboard architectures [32]. Procedural knowledge is encoded in a set of Knowledge Sources (KSs) similar in form to those of a blackboard system: each KS has a set of class-constrained variables and a set of parameterized preconditions which must all be true for the KS to be "applicable" to the state of the world model. Each KS also has a set of routines that are run if the KS is actually "fired," or chosen and executed.

The interpreter that chooses the next KS to fire is drawn almost directly from the PRS architecture [20], and bears little resemblance to a blackboard's agenda mechanism. Figure 7 shows the Lisp code for the prototype AIS interpreter. Each cycle of the interpreter finds the set of KSs whose preconditions are true in the current state of the world model (the "set of applicable KSs" or **soak**), and then asserts the value of the current **soak** into the world model, essentially making the world model represent the fact that the system is considering executing those KSs. This new assertion may cause new, meta-level KSs to match the world model state. The meta-level KSs are responsible for choosing which KS to fire from the **soak** matched in the previous interpreter cycle. Figure 8 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 world model 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,

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

Figure 8: The **strategy-choice** meta-level KS.

the system executes a single KS, chosen randomly from the **soak** of the previous reasoning level.

Our prototype AIS 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. 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.

Messages to the prototype AIS from the RTS are received into a buffered I/O process that interrupts the AIS when a complete message has arrived. The interrupt handler asserts the new message into the AIS world model and aborts the current activity. Control returns to the start of the interpreter loop, so that the AIS can then decide whether to process the incoming data, restart the previous task, or switch to a new task entirely. We are investigating ways in which the communication interrupt handler can save the state of a KS rather than aborting it [16].

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 [20] 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.

The prototype AIS has KSs to incrementally form hierarchical plans to navigate through the hallways of our floor. Given a destination and the current location of the Hero, KSs at the "hallway" abstraction level build a plan from hallway-path tasks. Hallway-path tasks are decomposed into one or more tasks at the "traversal" abstraction level. This decomposition may involve splitting the hallway into different sections which are traversed in different ways. For example, a long hallway might be traversed in a "ballistic-traverse" task for most of its length, so that the RTS only uses the Hero's wheel-encoders to keep track of its progress down the hallway. Then, when the end of the hallway draws near, the traversal can be switched to a "cautious-traverse" task, during which the Hero also checks its position and orientation by sensing landmarks (walls, doorways, and corners). At the traversal abstraction level, several KSs match each class of traversal task to suggest the TAPs needed to implement the traversal. These KSs essentially form the AIS' implicit reasoning about the graph model corresponding to the traversal phase and the reactions required to preempt

```
KS suggest-stop-if-object-ahead
   :VARIABLES (traversal-task-p ?task)
   :PRECONDITIONS (null (tap-list ?task))
   :ACTIONS (push (make-tap 'stop-if-object-ahead
                            :MAX-PERIOD (/ *safety-distance* (speed ?task)))
                  (tap-list ?task))
```

Figure 9: The **suggest-stop-if-object-ahead** KS.

failures. For example, the **suggest-stop-if-object-ahead** KS, shown in Figure 9, matches all classes of traversal tasks and adds the **stop-if-object-ahead** TAP to the list of suggested TAPs for the task. Note that the MAX-PERIOD for the TAP is based on the speed at which the traversal will take place. When the **strategy-choice** meta-level KS chooses to decompose a traversal task, it executes *all* of the KSs which suggest TAPs for the traversal task, rather than choosing just one.[3] The resulting list of TAPs is then sent to the Scheduler.

5.2 The Prototype Scheduler

In the current implementation, the RTS can run only one TAP at a time, and TAPs are not interruptible, so the prototype Scheduler does not need to consider TAP preemption. For the sake of simplicity, the Scheduler also assumes that TAPs are independent of each other: that is, the firing of one TAP has no bearing on the firing of any other TAP. In fact, this is not true, since sets of TAPs can enable each other or act in a mutually exclusive fashion. For example, if one TAP specifies (`*moving*`) in its precondition and another TAP specifies (not `*moving*`), the TAPs cannot both fire in a cycle of the RTS unless an intermediate TAP changes the value of `*moving*`. A more sophisticated Scheduler would be able to take into account such dependencies and produce more efficient schedules.

By assuming TAP independence and TAP atomicity, the prototype Scheduler can use a simplified deadline-driven scheduling algorithm [30] to optimally derive 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 closest deadline. To derive a cyclic schedule with this criterion, the Scheduler simulates the operation of a dynamic scheduler, incrementing a time counter and deciding which TAPs to run as simulated time passes. After the simulation has progressed far enough to invoke the TAP with the maximum MAX-PERIOD, the Scheduler begins scanning the trace of the simulation, attempting to extract a loop of TAP invocations which meets all TAP timing requirements. The maximum possible loop size is equal to the least common multiple of the TAP MAX-PERIODs. Note that we can relax the atomicity requirements, and allow TAP preemption, without losing the ability to build a provably optimal schedule. Deadline-driven scheduling has been shown to be optimal for such problems [30], but the short execution time of current TAPs does not warrant the cost of adding preemption mechanisms to the RTS.

If the Scheduler cannot build a successful schedule to guarantee all the TAP timing constraints, it returns a failure message to the AIS, which must adjust its proposed TAP list to loosen the scheduling constraints. The prototype AIS can make this adjustment by altering the traversal task

[3] Figure 8 shows a simplified version of the **strategy-choice** meta-level KS that always invokes one KS.

```
1    AIS: Decomposing hallway task #<TASK HALLWAY-PATH1>
2    AIS: Creating new cautious traverse #<TASK CAUTIOUS-TRAVERSE3>
3    AIS: Speed = 16 inches/second, Distance = 427 inches, Deadline = 36 seconds
4    AIS: Running all KSs for strategy SUGGEST-TAPS on #<TASK CAUTIOUS-TRAVERSE3>
5    AIS: Suggesting check-orientation TAPs, max period 1.88 seconds
6    AIS: Suggesting get-next-schedule TAP
7    AIS: Suggesting follow-hall TAPs
8    AIS: Suggesting stop-if-object-ahead TAP, max period .50 seconds
9    AIS: Running Scheduler for #<TASK CAUTIOUS-TRAVERSE3>...
10   SCHED: FAILURE: #<BEHAVIOR STOP-IF-OBJECT-AHEAD> exceeded max period of .50
11   SCHED: WARNING: No schedule was possible
12   AIS: Decreasing speed for #<TASK CAUTIOUS-TRAVERSE3>
13   AIS: Set speed to minimum required to meet deadline:  12 inches/second
14   AIS: Running all KSs for strategy SUGGEST-TAPS on #<TASK CAUTIOUS-TRAVERSE3>
15   AIS: Suggesting check-orientation TAPs, max period 2.50 seconds
16   AIS: Suggesting get-next-schedule TAP
17   AIS: Suggesting follow-hall TAPs
18   AIS: Suggesting stop-if-object-ahead TAP, max period .67 seconds
19   AIS: Running Scheduler for #<TASK CAUTIOUS-TRAVERSE3>...
20   SCHED: Successful schedule is:
21   SCHED:       #<BEHAVIOR CHECK-ORIENTATION>
22   SCHED:       #<BEHAVIOR STOP-IF-OBJECT-AHEAD>
23   SCHED:       #<BEHAVIOR FOLLOW-HALL>
24   SCHED:       #<BEHAVIOR STOP-IF-OBJECT-AHEAD>
25   SCHED:       #<BEHAVIOR GET-NEXT-SCHEDULE>
26   SCHED:       #<BEHAVIOR STOP-IF-OBJECT-AHEAD>
27   SCHED:       loop.
```

Figure 10: A trace of AIS/Scheduler interactions. "Behaviors" consist of one or more TAPs grouped together for scheduling efficiency. This trace does not illustrate any TAPs on the unguaranteed list.

and then re-running the KSs which suggest the TAPs for the task. The KSs adjust their parameters based on the new traversal task, and create an alternative TAP list. Our implementation currently has two methods for altering a traversal task to ease the scheduling problem. Figure 10 shows a system trace illustrating the first method, in which the AIS changes a traversal parameter such as the Hero's speed. In the example, the AIS initially proposes to use a 16 inch/second "cautious" traverse (lines 1–3), and KSs suggest the appropriate TAPs with the necessary timing constraints (lines 4–8). The Scheduler is unable to build a schedule (lines 9–11), so the AIS decreases the traversal speed to the minimum required to meet the traversal deadline (lines 12–13), and deletes the previous TAP suggestions. KSs then fire to suggest TAPs with revised timing constraints (lines 14–18), and this time the Scheduler succeeds in building a cyclic schedule.

The AIS' second method for ameliorating scheduling problems is to change the class of the traversal. For example, changing a "cautious" traversal to a "ballistic" traversal eliminates the need to guarantee the TAPs that sense landmarks[4] and check orientation, and thus decreases the number of TAPs to be scheduled. Ballistic-traversal tasks actually include the landmark-sensing TAPs on the unguaranteed list, so that if the RTS has time it will try to verify the Hero's position. Thus changing the class of a traversal task can have the effect of moving a TAP from the guaranteed to the unguaranteed TAP list, or dropping the TAP entirely.

6 Comparison to Related Work

In recent years, numerous systems have been developed to apply AI techniques to real-world domains, particularly those which involve response deadlines. In this section, we discuss related research from an architectural point of view, where the form of the system and the division of responsibilities among its components is significant. We will also consider the types of performance guarantees these systems can provide, and thus how well they address real-time control issues. There are two common approaches to developing intelligent real-time control systems: embedding an AI system within a real-time system, and vice versa [11].

6.1 Embedding Intelligence in a Real-Time System

Most researchers strive to embed intelligence within a real-time system, so that the AI mechanisms are required to meet deadlines. 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 [24, 27]. Unfortunately, this approach engineers out of the AI system the high-variance 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.

Another approach to embedding intelligence in a real-time system assumes that the system runs on a single processor which must satisfy both the bounded real-time tasks and the uncertain AI tasks. Paul *et al.* [33] deal with this problem by encapsulating high-variance AI tasks within a server process that is scheduled to run around the real-time tasks. Thus the real-time tasks can be guaranteed to meet their deadlines, and the AI task assumes the remaining CPU time (minus context switching overhead). Kaelbling [21] presents a similar approach in which the real-time processes are described in REX, a language with provable run-time properties. While these systems achieve the desired isolation of the real-time processes from the AI processes, they do not address the fundamental issue of how the systems interact to provide overall goal-directed performance. Hendler and Agrawala [18] are integrating the Dynamic Reaction system and the MARUTI operating system, to implement guaranteed real-time reactive reasoning in a manner very similar to CIRCA's guaranteed TAP schedules. They too are investigating the mechanisms by which a planning system can generate real-time task requirements. However, by restricting the system to a single processor, they exacerbate the complex issues of trading off action and deliberation [14, 29].

A refinement of the single-processor approach uses iterative improvement algorithms to guarantee that the intelligent system can be interrupted at any time and will still yield a solution, possibly with reduced precision or confidence [4, 19]. Such "anytime" algorithms cannot provide

[4] Sets of related TAPs can be grouped together into "behaviors" for scheduling efficiency. The **follow-hall** behavior in Figure 10 includes TAPs to recognize landmarks during a cautious traverse.

any performance guarantees, since the degree of response quality reduction is not under the system's control. The imprecise computation method [7] specifies that some minimum amount of computation is performed before the problem-solving can be interrupted, and thus this method could make guarantees about its worst-case performance.

However, all of these methods are limited to problems which are suited to iterative improvement algorithms. In many real-time applications, meeting time constraints might require producing responses that are significantly different from the responses that would be appropriate without time constraints. And, in many situations, approximate responses are not desirable at all. For example, when traversing a crowded area, a quick approximation (to an ideal collision avoidance maneuver) such as "turn right 70 degrees, plus or minus 30 degrees" might lead to disaster, while a completely different response such as "halt" would be far better [11].

6.2 Reactive Architectures

At the other end of the "processor count" dimension, embedding intelligence in a real-time system leads to reactive architectures such as Pengi [1] and the subsumption architecture [5]. Like CIRCA, these systems have separate behaviors which are each responsible for recognizing and reacting to specific input patterns. However, these systems assume that all of their behaviors are running concurrently on separate processors. In addition to the scaling problems discussed earlier, the processor-per-behavior scheme also wastes computing power, since many behaviors need not be active at various times. Other systems [8, 39] have made provisions to activate only subsets of behaviors, in much the same way as CIRCA runs different TAP schedules. However, these other 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[5]. CIRCA's resource allocation and scheduling are crucial to the system's flexibility, extensibility and efficiency. And, 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" [24].

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 [14]. 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.

6.3 Embedding Reactivity in an AI System

Other researchers 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. Soar [26], for example, encodes reactive knowledge in the same production form as its other knowledge, with the added feature that reactive operators are fired immediately after they are matched [25]. This eliminates the possibility of recursive subgoaling, providing an immediate reaction once the productions are matched. However, since Soar tries to match all its productions all the time, its growing knowledge base makes bounding

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

reaction time problematic. Assumptions of complete match parallelism lead to the same scaling difficulties and inefficiency as the reactive architectures discussed above.

CIRCA avoids these problems by choosing the subsets of reactive knowledge it will test during each cycle of the RTS. These choices prevent CIRCA from displaying the 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 [25].

6.4 Cooperative Systems

CIRCA demonstrates an alternative to the embedded approaches, using separate, concurrent AI and real-time subsystems to cooperatively produce the desired performance. Hanks and Firby [14] are investigating a similar approach, combining a transformational planner [15] with an execution module based on Reactive-Action Packages (RAPs) [12]. Their work does not concentrate on providing timeliness, since RAPs are executed by a complex queue manager. Also, the strategic planning and RAP execution subsystems share a global world model; this shared resource could lead to contention problems that would delay the subsystems. CIRCA avoids shared data for this reason, and relies instead on message passing and interrupts.

Arkin's Autonomous Robot Architecture (AuRA) [2] includes a reactive execution subsystem and a hierarchical planner that determines which reactive "schemas" are active. A world modelling subsystem controls AuRA's stored knowledge, providing an interface that avoids shared-memory assumptions. 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's AIS and Scheduler can provide similar functionality by monitoring the RTS, as mentioned in Section 4.5. AuRA does not address the timeliness or resource restrictions that are the focus of our architecture.

Simmons' Task Control Architecture (TCA) also combines reactive and planning systems [35, 37]. 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 [36].

Miller and Gat have developed the three-layer ATLANTIS system [31], 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.

7 Current Status & Future Directions

As presented in Section 5, the prototype AI subsystem successfully plans hallway paths, breaking them into traversal phases and developing TAP schedules in cooperation with the prototype Scheduler. Currently, the Scheduler is encapsulated within a KS, and returns either a successful schedule or `nil`. We are investigating 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 suggested set of TAPs. This will allow the AIS and Scheduler to interact in a more intelligent manner than the current "generate and test" scheme. We are also examining more powerful scheduling algorithms that would account for dependencies between TAPs or a multiprocessor RTS.

The RTS, described in Section 4.4, has successfully piloted the Hero through numerous hallway traversals, including stopping for obstacles, rounding corners, and recognizing doorways. Although the Hero's inaccurate sensors and wheel slippage have caused occasional navigation failures, the safety of the robot has not been violated (it has never collided with walls or obstacles). We are planning to re-implement the RTS in C, to eliminate Lisp's unpredictable garbage collection and to speed performance.

We also seek to evaluate and verify the prototype system's timeliness guarantees. The graph model and the concept of a safe set of states are the foundation for the architecture's control-level performance guarantees. The guarantees which an implementation makes depend on the accuracy of the timing information used to schedule TAPs and anticipate control-level deadlines. Currently, the AIS and Scheduler use fixed, estimated timing parameters. We hope to develop methods which can derive the relevant domain deadlines from a model of the environment, providing a sound basis for safety guarantees and the adjustment of TAP parameters.

In summary, CIRCA is an innovative architecture in which cooperating subsystems provide both the performance guarantees needed for real-time control and the powerful, unpredictable intelligence needed to address complex task-level problems. Using deadline-driven scheduling and a graph model of the RTS and the environment, CIRCA represents a unique and promising method of applying unrestricted AI techniques to tasks with hard real-time deadlines.

Acknowledgment

The authors would like to thank Marcel Schoppers and the anonymous reviewers for many insightful comments on this work.

References

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

[3] J. E. Arnold, "Experiences with the Subsumption Architecture," in *Conf. on Artificial Intelligence Applications*, pp. 93–100, 1989.

[4] M. Boddy and T. Dean, "Solving Time-Dependent Planning Problems," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 979–984, August 1989.

[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] J.-Y. Chung, J. W. Liu, and K.-J. Lin, "Scheduling Periodic Jobs That Allow Imprecise Results," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1156–1174, September 1990.

[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 D. McDermott, "Temporal Data Base Management," *Artificial Intelligence*, vol. 32, no. 1, pp. 1–55, April 1987.

[10] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, no. 3, pp. 231–272, 1979.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] B. Hayes-Roth, "A Multi-Processor Interrupt-Driven Architecture for Adaptive Intelligent Systems," Technical Report KSL 87–31, Knowledge Systems Laboratory, Stanford University, June 1987.

[17] 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.

[18] 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.

[19] E. J. Horvitz, "Reasoning About Beliefs and Actions Under Computational Resource Constraints," in *Proc. Workshop on Uncertainty in AI*, 1987.

[20] F. F. Ingrand and M. P. Georgeff, "Managing Deliberation and Reasoning in Real-Time AI Systems," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 284–291, November 1990.

[21] L. P. Kaelbling, "An Architecture for Intelligent Reactive Systems," in *Proc. Workshop on Reasoning About Actions and Plans*, pp. 395–410. AAAI, 1986.

[22] J. A. Koomen, "The TIMELOGIC Temporal Reasoning System," in *University of Rochester Computer Science Department Technical Report 231*, 1989.

[23] R. E. Korf, "Real-Time Search for Dynamic Planning," in *Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[24] 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.

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

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

[27] 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.

[28] V. R. Lesser, J. Pavlin, and E. Durfee, "Approximate Processing in Real-Time Problem Solving," *AI Magazine*, vol. 9, no. 1, pp. 49–61, 1988.

[29] R. Levinson, "Autonomous Prediction and Reaction with Dynamic Deadlines," in *Proc. AAAI Spring Symp. on Planning in Uncertain, Unpredictable, or Changing Environments*, 1990.

[30] 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.

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

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

[33] 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.

[34] D. W. Payton, "An Architecture for Reflexive Autonomous Vehicle Control," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, volume 3, pp. 1838–1845, 1986.

[35] 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.

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

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

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

[39] 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.

[40] 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.