PLAN GENERATION AND HARD REAL-TIME EXECUTION WITH APPLICATION TO SAFE, AUTONOMOUS FLIGHT

by

Ella Marie Atkins

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
1999

Doctoral Committee:

Associate Professor Edmund H. Durfee, Co-Chair
Professor Kang G. Shin, Co-Chair
Professor Daniel E. Koditschek
Professor N. Harris McClamroch
Associate Professor Michael P. Wellman

To Deano and Lucas.

ACKNOWLEDGEMENTS

I have had the distinct pleasure of working with several research groups while at the University of Michigan, including the AI and real-time system labs within EECS and the UAV lab in the Aerospace Engineering department. I have learned a great deal from each group that I expect to be quite useful as I embark on my future career in academia.

Most students are lucky to find one good advisor. I have been fortunate to have two, Professors Ed Durfee and Kang Shin, and I am very grateful for all their support, ranging from financial to research ideas to dissertation help. It is not all students that find themselves in a situation where they have to specifically request that they not be funded so they are allowed to teach for a semester.

During my work on the CIRCA project, I have been privileged to collaborate with two other graduate students, Tarek Abdelzaher and Chip McVey. Tarek is one of the hardest-working students I have ever known, which combined with his zeal for academic research and personable nature will make his future employer very lucky . We have spent many hours discussing how to get planners and schedulers to really communicate, and I owe many of my best ideas to these discussions. Chip was drawn to the CIRCA research group as he strayed from the world of physics and was the perfect collaborator for CIRCA research. I very much missed him after he departed Michigan for the money-laden world of the computer industry; perhaps I should have tried harder to bribe him into staying at Michigan for his Ph.D.

I inherited the original CIRCA architecture from David Musliner after he graduated. Although he and I have not always had exactly the same ideas about the “best” algorithms for CIRCA, he has always provided a different and thoughtful view on

To Deano and Lucas.

ACKNOWLEDGEMENTS

I have had the distinct pleasure of working with several research groups while at the University of Michigan, including the AI and real-time system labs within EECS and the UAV lab in the Aerospace Engineering department. I have learned a great deal from each group that I expect to be quite useful as I embark on my future career in academia.

Most students are lucky to find one good advisor. I have been fortunate to have two, Professors Ed Durfee and Kang Shin, and I am very grateful for all their support, ranging from financial to research ideas to dissertation help. It is not all students that find themselves in a situation where they have to specifically request that they not be funded so they are allowed to teach for a semester.

During my work on the CIRCA project, I have been privileged to collaborate with two other graduate students, Tarek Abdelzaher and Chip McVey. Tarek is one of the hardest-working students I have ever known, which combined with his zeal for academic research and personable nature will make his future employer very lucky . We have spent many hours discussing how to get planners and schedulers to really communicate, and I owe many of my best ideas to these discussions. Chip was drawn to the CIRCA research group as he strayed from the world of physics and was the perfect collaborator for CIRCA research. I very much missed him after he departed Michigan for the money-laden world of the computer industry; perhaps I should have tried harder to bribe him into staying at Michigan for his Ph.D.

I inherited the original CIRCA architecture from David Musliner after he graduated. Although he and I have not always had exactly the same ideas about the "best" algorithms for CIRCA, he has always provided a different and thoughtful view on my work that has clearly assisted my research efforts. I believe the CIRCA research project will thrive well after I leave Michigan due to the continued support from Dave, as well as future generations of CIRCA students, including Haksun Li, Jonathan Arnold, and Jeremy Shapiro, each of which has already contributed to the CIRCA-II software.

After three years at Michigan, I could no longer resist returning to my Aerospace roots and began "playing with hardware" in the form of an Uninhabited Aerial Vehicle (UAV). I met two excellent graduate students, Robert Miller and Tobin VanPelt, who had taken the project under their wing, and I was fortunate to become a part of the UAV team. Our UAV would not exist were it not for the countless hours Tobin has spent building the physical aircraft structure and the time Robert has spent on instrumentation, managing all project details, and more recently, venturing into writing the software I haven't had time to complete due to this dissertation. Real progress on our UAV began when we coerced Keith Shaw, R/C modeler extraordinaire, to help us. Keith has volunteered countless hours and has been invaluable during the design, construction, and flight phases. His veritable addiction to and excitement for R/C flight has been contagious to the extent that I now find myself searching for new excuses to fly model airplanes for research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER I

INTRODUCTION

The goal of this dissertation is to introduce techniques for reasoning within an integrated plan generation and execution system and to apply 0these methods to the problem of safe, fully-autonomous operation within a complex real-time domain. Such a system has *limited computational resources* and may be required to utilize *imprecise knowledge* to build potentially *incomplete plans* that must *execute in hard real-time*. In this thesis, we describe methods for handling these constraints during plan development and execution. We focus on mechanisms to detect and respond in a timely fashion when the environment deviates into an area not handled within an executing plan, and explicitly make the tradeoffs required to ensure that each plan will not overutilize resources when executed, thereby guaranteeing avoidance of all (modeled) catastrophic failure modes.

Motivation

Autonomous behavior in complex real-world systems requires accurate and timely reactions to environmental events. These reactions must prevent all catastrophic failures such as loss-of-life and should ultimately achieve mission goals such as arriving at a destination on-time. Timely and accurate responses for a complex domain may require a significant amount of computational resources, regardless of whether such responses are pre-programmed or dynamically selected as the agent acts within its environment. As processor speed and algorithm efficiency increase, it is tempting to presume that resource limitations are not an issue because they can always be combated with a bigger, faster system. However, the exponentially-complex search-based planning and scheduling algorithms typically utilized to impart "intelligence" to a complex autonomous system can quickly consume all such resources, as can the storage and retrieval-time requirements for reactions in strictly plan-execution systems. Additionally, hardware upgrades are not easily performed in unfriendly, resource-limiting environments (e.g., space, underwater).

If neither dynamic planning nor plan-retrieval systems can alone be expected to respond to the spectrum of situations that may be encountered, then one alternative is to combine the two techniques, which has been done in several hierarchical architectures that are often mapped to a generic 3-tier [17] conceptual framework illustrated in Figure 1-1. We adopt such a multi-layer concept in this work. Our top deliberation level reasons about guaranteed real-time failure avoidance while building plans, then those plans execute in hard real-time on the plan-execution layer, with specific directives included to recognize and react as the system progressively deviates from the nominal environment for which the plan was constructed. Our "reactive" layer is comprised of the low-level algorithms required to interact with a [possibly continuous-time] complex environment. Overall, this design results in a flexible, hard real-time execution system that exhibits graceful performance degradation when computational resources are overloaded.

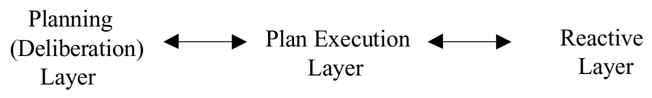


Figure 1-1: Three-tier (3-T) Architecture Concept.

Maintaining close ties with an application domain ensures that our algorithms have practical use and, in parallel, addresses key automation issues within that specific application. In this thesis, we ground our discussions with examples and challenging problems associated with achieving safe, fully-automated commercial aircraft flight, in which the "simple" overall goal is to take off and safely fly to some destination airport. In order to fully automate the tasks currently performed by the cockpit crew, the system must be capable of analyzing and responding to a diverse set of sensory data input, ranging from atmospheric conditions to other air traffic to detectable aircraft system failures. Additionally, flight is inherently hard real-time: no "indefinitely safe" state set exists once the aircraft leaves the ground because it cannot simply "stop in mid-air" while planning its next course of action. To tackle such a problem, a system must be capable of reasoning about a large array of complex environmental features to select reactions for each potential in-flight situation and also compute and enforce associated real-time deadlines so that the plane will not crash before the need to act was even detected.

Fortunately, techniques for automatic flight control are well developed and may be utilized versus redesigned when taking the final steps to fully automate the cockpit. Current flight management systems [44],[64] are capable of automatically flying an aircraft from takeoff through landing, with the cockpit crew dialing in course/destination changes and monitoring the progress of flight. Thus, if considering only nominal situations, full automation is possible today. However, such a system cannot be considered "safe" until it reacts to all dangerous situations for which the nominal plan is insufficient.

As a simple example, consider a situation in which adverse weather conditions arise at the destination airport. Once the problem has been identified, an appropriate contingency response is to automatically re-route to an alternate airport, thereby averting the problem. This specific behavior may easily be built into existing flight management systems; however, devising a comprehensive set of such reactions is much more difficult. For example, consider a more time-critical and low-probability event, such as the situation that occurred near Sioux City, Iowa, when engine parts violently flew off the aircraft and severed all hydraulic lines. The required response to such a situation is specific to the current aircraft state (e.g., altitude, bank/pitch angle, terrain, fuel, and aircraft controllability parameters). Furthermore, this situation would have been difficult to consider possible in advance because redundant backup systems were present and no such situation had previously been recorded. By handling imprecise knowledge and incomplete plans, we will illustrate how our architecture can be used to detect and respond to such unlikely and dangerous situations, and also how such safety-critical reactions will occur in time to avert an aircraft crash.

What's in a Plan?

For AI researchers, the term plan may refer to either an action sequence or else a policy that applies to a group of world states. Due to our veritable obsession with hard real-time plan execution, our plans must include more than constructs for matching actions to states. This section is devoted to disambiguating the definition of "plan" that we will presume throughout this dissertation.

Figure 1-2 illustrates two of the most popular interpretations of a plan. Figure 1-2a shows a STRIPS [18] plan, a traditional format produced by numerous state-space and plan-space planners. This specification is appropriate when actions must be strictly executed in a predefined sequence. The STRIPS plan structure does not rely on active sensing during plan execution, implying there can be no uncertainty about when or in what order actions should execute. Figure 1-2b illustrates a policy representation such as that generated by a traditional Markov Decision Process (MDP) [11]. In this model, there is uncertainty regarding the exact progression of states that will be encountered, so the set of current state features must be sensed and matched to the correct action to execute next. As a result, reaction times to environmental events are a function of the total time required to identify the current state, find the appropriate action, then execute that action.

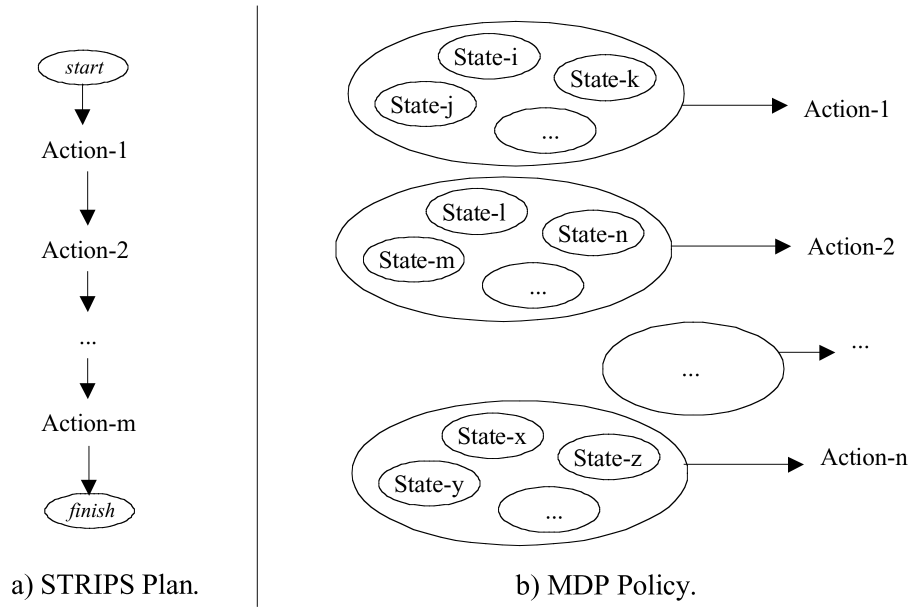


Figure 1-2: Traditional Plan Types.

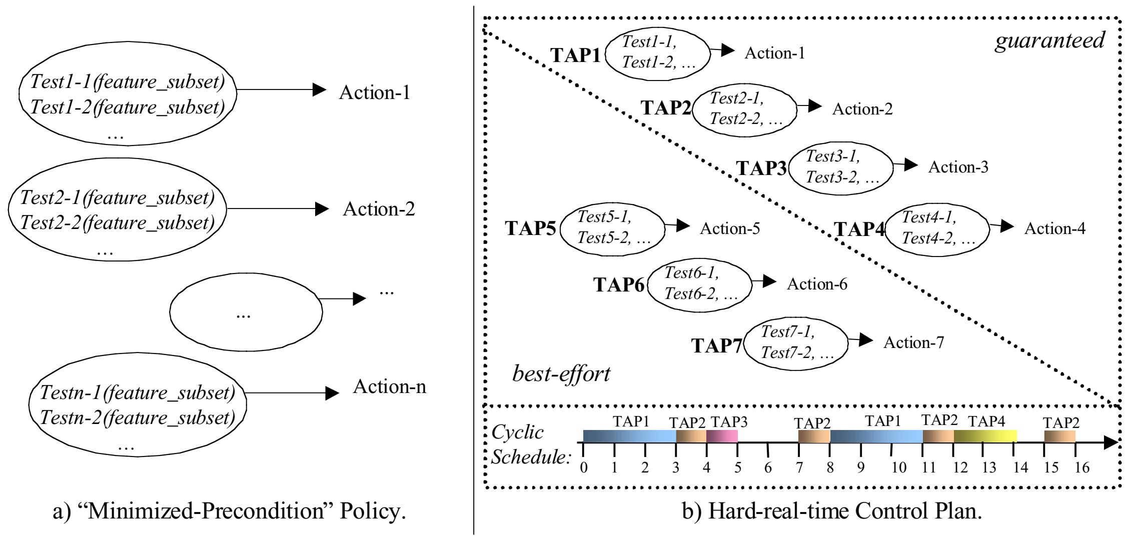

Test1-1(feature_subset)
Test1-2(feature_subset)
...
Action-1
Test2-1(feature_subset)
Test2-2(feature_subset)
...
Action-2
...
...
Testn-1(feature_subset)
Testn-2(feature_subset)
...
Action-n
a) "Minimized-Precondition" Policy.
guaranteed
TAP1 Test1-1, Test1-2, ... Action-1
TAP2 Test2-1, Test2-2, ... Action-2
TAP3 Test3-1, Test3-2, ... Action-3
TAP5 Test5-1, Test5-2, ... Action-5
TAP4 Test4-1, Test4-2, ... Action-4
TAP6 Test6-1, Test6-2, ... Action-6
TAP7 Test7-1, Test7-2, ... Action-7
best-effort
Cyclic Schedule:
TAP1 TAP2 TAP3 TAP2 TAP1 TAP2 TAP4 TAP2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b) Hard-real-time Control Plan.

Figure 1-3: Evolution of the Real-time Control Plan.

Because we allow uncertainty about the progression of world states, we must sense state features to select appropriate actions during plan execution. However, we also require that the complete sense-act loop execute in hard real-time for failure-avoidance purposes. To define our control plans, consider an MDP policy as the initial representation. Now, in order to increase efficiency for matching the current world state to an action, consider a new format in which the policy is post-processed so that a set of general "preconditions", not fully-instantiated states, is used to uniquely match each reachable state to a policy action. This "minimized-precondition" policy representation is shown in Figure 1-3a.[1]

In a policy where the exact sequence of states cannot be predicted, the "minimized-preconditions" for executing each action must be checked periodically, with each action executing whenever its preconditions match. Otherwise, the action may never execute in a state where it has been planned. This plan structure suggests a loop over the precondition-action pairs to identify and execute the proper action for each state. A cycle through the plan-loop will not execute instantaneously, so we must structure the plan so that each action's preconditions will be tested with sufficient frequency to guarantee avoiding any failures that might occur should action execution delay too long.

If all actions are required for failure-avoidance and all actions have the same real-time execution deadlines for failure-avoidance, then the best we could do is to cycle through the plan-loop as-is. However, typically, only certain actions are required for failure-avoidance while others are used only for goal-achievement. We attach to each action the worst-case timing requirements for guaranteed failure-avoidance, and classify all actions with specified worst-case timings as "guaranteed" while all others are "best-effort", as illustrated in Figure 1-3b. Now, if all guaranteed actions have the same worst-case timing requirements, we can execute the "plan-loop" over all guaranteed actions, inserting best-effort actions into slack time intervals when available. However, in general, the guaranteed actions may have a very diverse set of real-time requirements.

---

[1] In the worst-case, the minimized precondition will be equivalent to observing all state features. Generally, however, nontrivial feature-sensing activities can be reduced with the minimized-precondition format.

Thus, instead of looping over each action in the guaranteed set, we may maximize our ability to guarantee that all execute in time by explicitly scheduling these actions in accordance with their resource requirements and real-time deadlines.

Figure 1-3b includes a cyclic schedule that specifies the "plan-loop" for the set of guaranteed actions for this plan. We define a *task* as the combination of the minimized-precondition feature tests for the action as well as the action itself. For guaranteed performance, this schedule must be built assuming worst-case task resource consumption, and must verify that all real-time constraints for the associated action will be met during execution.

For this dissertation, we define a *plan* as the Figure 1-3b combination of a minimized-precondition task set and cyclic task schedule that guarantees real-time failure-avoidance during plan execution. Figure 1-4 shows the generic components required for generating and executing a real-time plan, including a planner, task scheduler, and plan executor with predictable worst-case task execution properties. During execution, highest priority will be given to timely execution of the guaranteed cyclic task schedule, then any lower-priority best-effort tasks will be dynamically inserted into any slack-time intervals, as is done in traditional real-time task execution models [37].

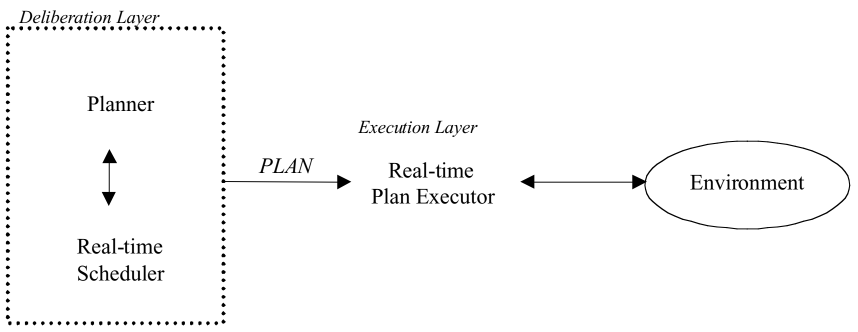


Figure 1-4: Architecture for Generating and Executing Real-time Plans.

Problem Statement

This dissertation addresses the problem of successfully constructing and executing these real-time control plans in a complex system with incomplete domain knowledge, limited resources, and a dangerous environment in which missing a deadline may be quite deadly. Such catastrophic system failure is typically avoided in other planning, scheduling, and plan execution systems by making restrictive assumptions about resource availability and knowledge base extent. We relax these assumptions to better reflect a real-world environment, but are then forced to perform tradeoffs in ensuring safety and goal completion.

Guarantees of safety and goal completion on a system with limited computational resources are problematic. Additionally, traditional planners assume sufficient memory and time will always be available, and thus do not even address resource limits. When a planner proposes a set of tasks with *real-time constraints* (i.e., deadlines), a scheduler may find it impossible to fit all tasks into the fixed set of plan-execution resources. One approach to this problem is to modify this set of tasks to reduce plan resource requirements, but traditional real-time schedulers are not able to reason about the environmental implications of such modifications. Our architecture adopts this approach by prioritizing safety over goal completion during planning, and by providing for clear communication between scheduler and planner to allow negotiation between the two when required. Our planner is thus able to intelligently direct tradeoffs necessary to satisfy scheduling requirements as well as ensure overall safety.

An absolutely safe plan may, however, still be incompatible with limited resources operating in a dangerous environment where hard real-time guarantees must be met to avoid failure. In this case, we may be required to relax absolute safety guarantees to *probabilistic guarantees,* especially in a domain with a variety of diverse failure modes having calculable or empirically derived probabilities. Our planner will thus incorporate flexible parameters, such as a probability threshold below which unlikely states are ignored, that can be dynamically modified to reduce scheduling requirements. Solving the problem of estimating the probabilities of various contingencies arising and establishing an appropriate threshold will allow graceful performance degradation even with severely constrained resources. The tradeoff associated with probabilistic guarantees is that an executing plan may stumble into a state that has been considered improbable, thus not planned for, and the system must be equipped to “find its way out”.

Incomplete domain knowledge generates a similar situation in which an executing plan may reach an unexpected state, i.e., one it had no knowledge of, from which it cannot continue. This incompleteness derives from two sources: First, plans are constructed using a *stochastic model* of world events, which is common for probabilistic planning algorithms but does not precisely specify the sequence of states that is expected to occur. Second, we allow for an *incomplete* domain description since this is often the case for real-world systems.[2] Such incomplete knowledge leads to incomplete plans, which in turn may result in reaching unexpected states during plan execution.

Encounters with these unexpected or unplanned-for states are addressed by requiring our system to *recognize* and *react* when the world (i.e., environment) enters a state not handled within the executing plan. Thus, for a resource-limited system, the problem is to design a system that knows its limits and devotes sufficient resources to watching for such situations; in a sense it must "expect the unexpected". Otherwise, catastrophic failure may occur without the system ever noticing (e.g., an office robot without knowledge of terrain may happily drive off a cliff). Reacting to such state ties together all components within the generic Figure 1-4 architecture, a version of which we adopt in this dissertation. In our system, we must build actions into each plan to initially

[2] We do, however, require a model for all events that lead quickly to catastrophic failure. Otherwise, we would not even be able to identify all dangerous situations.

recognize each anomalous state, then the planner and scheduler must advise the executor, via a new plan, of an appropriate reaction. This must be done in real-time if a failure-avoidance reaction is required.

Approach

We utilize a combination of planning, scheduling, and real-time plan execution algorithms to achieve safe, autonomous operation of a complex system. We place primary emphasis on failure-avoidance by computing and adhering to hard real-time deadlines that allow failure-avoidance guarantees. However, since we also emphasize practical issues associated with automating complex systems, we must allow for tradeoffs when it impossible to simultaneously guarantee safety along with all other mission goals. Thus, we place secondary emphasis on goal-achievement, requiring only best-effort (soft real-time) execution of tasks that are not safety-critical.

Because of the difficulties in simultaneously achieving *timeliness* and *accuracy* while performing search-based planning and scheduling operations, our system adopts the philosophy that the planner and scheduler should reason *about* real-time, and be carefully designed such that it need not be constrained to reason *in* real-time. Developed plans are specified so that, when executed, they will be guaranteed to meet all safety-critical hard real-time deadlines. This approach may be debated because in the worst case, a universal plan set [63] is infeasible to create [23] and even more difficult to execute under hard real-time constraints, thus any fixed set of reactive plans may be constrained such that dynamic updates (replanning) may be required, even to avoid catastrophic failure. Our system explicitly reasons offline about how to minimize the likelihood of catastrophic failures, not by assuming "indefinite safety" during execution of each plan, but instead by caching a group of contingency plans that are specifically designed to redirect the system into a temporarily-safe set of states (e.g., a holding pattern for an aircraft) that maximize the amount of time the planner has to replan (e.g., develop a new route to an alternate airport).

A key aspect of our approach is the consideration of imprecise knowledge during planning. We utilize a probabilistic model to represent world events, and additionally incorporate algorithms that allow recovery of the system even if this external event model is incomplete or incorrect. However, in order to allow any assurances of failure avoidance, we must place some restrictions on knowledge accuracy. Otherwise, system behavior may be random at best, destructive in the worst-case. In our system, we assume that all features are fully-observable, although a non-trivial [bounded] set of resources may be required to actively determine feature values. We also assume specified actions will always execute safely, given that they are only executed when their preconditions match the current state. In this manner, we allow imprecise knowledge about external events, but require precise knowledge about the system itself and its ability to act on the environment.

We did not develop a new architecture for this research, maximizing our ability to incorporate existing architectural capabilities as well as avoiding introduction of yet another member to the already-overpopulated "AI architectures" cauldron. Instead, we built upon the Cooperative Intelligent Real-time Control Architecture (CIRCA) [51],[53] which combines planning, scheduling, and plan execution to enable hard real-time system response for plan execution, thus already addressing many of the integrated planning/execution coordination issues demanded for a fully-autonomous system. We have added new capabilities and relaxed several restrictive assumptions (e.g., accurate, complete knowledge and "indefinite safety" during plan-execution), resulting in a "second-generation" system we aptly entitle "CIRCA-II".

Both CIRCA and CIRCA-II approach automation problems by requiring guaranteed failure-avoidance and giving secondary consideration to goal-achievement. Instead of directly addressing issues of optimality across the overall hybrid planning/execution system, CIRCA-II explicitly divides available computational resources into two distinct groups: planning and plan execution. For this work, the set of planner/scheduler computational resources is presumed to be sufficient, given that no real-time constraints on deliberation have been imposed.[3] Then, plan-execution resources are explicitly scheduled to allow hard real-time response for guaranteed failure-avoidance. In this manner, CIRCA and CIRCA-II are able to make failure-avoidance guarantees by constructing and scheduling plans offline that require hard real-time response, requiring online construction of new plans only if the original plan set does not accomplish all secondary goals (e.g., compute a route to an alternate airport after an immediate turn to avoid adverse weather that could jeopardize aircraft safety).

Although we do not explicitly limit planning resource usage, we have developed a temporal model for planning that minimizes resource usage via a cyclic state-space representation and an abbreviated state transition set. This temporal model uses time-dependent event probabilities to compute state probabilities during planning. Additionally, the planner determines plan-execution timing constraints from "dangerous" event parameters along with action worst-case execution times. The popular Markov Decision Process (MDP) planners [11] provide a framework for optimal policy construction using a probabilistic transition model, and our approximate temporal model used for CIRCA-II planning is often compared to this approach. Ideally, CIRCA-II would incorporate an MDP planner, and then plans would be optimal whereas CIRCA-II's current plans are only sufficient. However, the MDP becomes highly complex for our problem because it must be augmented to consider action deadlines and state history (path) information as is currently done within our planner. We describe the MDP model

[3] A crucial element of future development for CIRCA-II is to explicitly consider resource constraints within planning/scheduling themselves, particularly when real-time constraints for online deliberation must be imposed. Although no such algorithms have yet been fully developed, this topic is discussed in Appendix D of this dissertation.

in Chapter II, then revisit the comparison of our approach to the MDP model in Chapter IV of this dissertation.

In addition to constructing plans and determining their real-time constraints, the CIRCA-II planning system must also be able to make tradeoffs when a plan cannot be successfully scheduled to meet these constraints. We always bias the system to favor safety over goal achievement; thus we begin by enforcing real-time constraints exclusively for failure-avoidance tasks. Even with this bias, all preemptive tasks sets developed through exhaustive planner backtracking may still be unschedulable. In this case, the original CIRCA would simply fail. However, we utilize our probabilistic temporal model to relax real-time constraints such that absolute guarantees become probabilistic, thereby allowing a non-zero chance of failure, but certainly a better chance of success than if the planner simply "gave up" because absolute guarantees were impossible to achieve.

Whenever a given set of actions cannot be scheduled, the set of actions must be modified. Because the planner and scheduler perform different functions, it is not straightforward for the scheduler to specify information that will guide the planner toward a schedulable plan. We first approached this problem by testing the ability of the planner to estimate its probabilistic guarantee requirements based on overall processor utilization, and have further adopted a heuristic cost-function approach so that the planner is directed to modify or replace specific "bottleneck" tasks during backtracking. We continue work on this problem to improve our heuristic for selecting bottleneck tasks, and have begun work to identify how the scheduler may alter task Quality-of-Service (QoS) levels to make tradeoffs that complement planner backtracking to alter the planned task set.

Once the set of carefully-scheduled plans have been developed, the plan-execution system must execute them such that all specified real-time constraints are met. We allow execution of incomplete plans, where "incomplete" means there may exist reachable state(s) that are not handled within the executing plan. For some "unplanned-for" states, the system may be able to continue blindly executing its plan without compromising safety. In other states, the system may need to recognize it has deviated from the planned-for state set and request help before safety is compromised. To do this in CIRCA-II, we require that the planner identify and build tests to detect important "unplanned-for" states, and that the execution system must be able to recall prebuilt plans to react to these states in hard real-time when necessary.

Contributions

The contributions of this thesis are directed toward the development of algorithms for the CIRCA-II architecture and demonstration of their use for enhancing safety in fully-automated domains. However, many of these algorithms and systems have general application to the AI, real-time, and aircraft automation communities. For the *probabilistic planning* community, this thesis presents a novel discrete-time probabilistic planning approach that requires substantially fewer resources than an MDP-based system at the cost of plan optimality. From a *multi-layer architectures* perspective, this thesis describes how a system can automatically build tests for and subsequently react to "unhandled" states, in real time when required. CIRCA-II must incorporate an *integrated planning-scheduling system* to enable real-time guarantees. This thesis also explores efficient and expressive communication protocols between distinct AI planning and traditional real-time scheduling algorithms to assist with the tradeoffs required when scheduling a proposed plan does not succeed. Finally, this thesis studies the application of all CIRCA-II techniques toward the ultimate goal of *safe, fully-automated aircraft flight*, and describes the flexible simulation and actual hardware testbeds that have already become an integral part of funded industrial research efforts. We describe pertinent details of each contribution in terms of CIRCA-II below.

Probabilistic Planning

We have developed and incorporated into CIRCA-II a temporal model that represents the state-space probabilistically and compactly, allowing for time-dependent state transition probabilities and non-trivial feature observation delays. As in CIRCA, we maintain the ability of the planner to compute preemptive task requirements for guaranteed failure avoidance. Additionally, we incorporate the ability to identify and ignore low-probability states when scheduling fails, enabling graceful performance degradation by trading off absolute failure-avoidance guarantees in favor of probabilistic safety guarantees when required.

Our temporal model uses a discrete time representation to describe transition and state probabilities, along with a cyclic state-space representation to minimize planning memory and time requirements. This model can show substantial representational and computational efficiency gains over Markov-based models but to-date contains no principled methodology for measuring solution optimality. Although not perfect, this introduces an alternative to the MDP community, which we hope will as a minimum rekindle discussions of when to use state-space planners rather than MDPs for complex, probabilistic problem domains.

Multi-layer Architectures

Incomplete plans are inevitable with imprecise knowledge and low-probability states that are either ignored or unmodeled. A plan-execution system cannot be expected to automatically "know" when it has deviated from the "planned-for" state set, so we have augmented the CIRCA-II planner such that it builds active perception tasks into each plan to detect crucial "unplanned-for" states, including those that may ultimately lead to catastrophic system failure. The planner also builds contingency plans to handle the "dangerous" subset of these unplanned-for states in real-time should they actually be encountered, and is capable of dynamic replanning when the plan-execution system feeds back an unplanned-for state that has not been completely handled by an existing contingency plan.

We have integrated a plan cache into the plan-execution layer of CIRCA-II. This cache is part of a "plan dispatcher" that is responsible for managing plan storage and controlling the execution of individual plans by monitoring the current world state and matching this state to the cached plan set for both "goal-oriented" and "contingency" plans required to maintain system safety. The dispatcher manages all communications from the planner and state feedback generated within an executing plan, such that new plans are added to the cache as they arrive and retrieved in hard real-time as they are required.

Planner-Scheduler Negotiation

The CIRCA-II planner proposes a set of tasks with execution constraints to be scheduled. In the original CIRCA, if the scheduler was unsuccessful, it fed back a non-descript "fail" message to the planner, which then blindly backtracked in an attempt to find a schedulable set of tasks. For this thesis, we directed our research efforts toward increasing the expressivity of scheduler feedback to help guide the planner when scheduling fails. Cooperative work in [47] illustrates how overall processor utilization can be used to guide planning during backtracking when all plan-execution occurs on a single processor. In follow-on work, we have developed a method by which utilizations from a multi-resource scheduler may be heuristically combined with planning data to select "bottleneck" tasks for guiding the planner during backtracking. Additionally, we have proposed a method for developing a set of fault-tolerant plans, allowing the planner to alter the content of plans when computational resources fail instead of just rescheduling them on available resources as is traditionally done in the fault-tolerant scheduling field.

Safe, Fully-Automated Flight

No grand-scale claims of "safe, fully-automated flight" can realistically be made, but we have begun to address crucial issues that have not been considered within flight management systems. We utilize CIRCA-II to perform "pilot-oriented" decision-making tasks automatically and in real-time. Such decisions include identifying and acting on conditions that necessitate a go-around during landing, or modifying the flight plan appropriately when significant airframe icing is detected. For this research, we primarily utilize the "flight domain" for demonstrating the operation of our CIRCA-II algorithms. However, to do this, we have become very much involved with the details of achieving automated flight, both from the "simulated" and "real" aircraft perspective.

The primary testbed for CIRCA-II algorithms has been the Aerial Combat Maneuver (ACM) F-16 simulator [58] that runs under Windows and on most UNIX platforms. We have built a low-level controller that interfaces with CIRCA-II, and added several "interesting" keyboard-driven events to the simulated environment. We wrote a planner knowledge base that allows fully-automated flight along with response to a number of simulated emergency situations.

We have also worked to implement CIRCA-II on an actual aircraft, the University of Michigan Uninhabited Aerial Vehicle (UAV). Research has included the development and implementation of a real-time software architecture for the aircraft running under the QNX real-time operating system. Additionally, we have implemented CIRCA-II in QNX and integrated it into the software for mission planning and fault recovery operations. We are currently implementing state estimation, control, and fault detection software for our UAV so that CIRCA-II can be used to fully-automate the aircraft during test flights which we expect to begin within the next six months.

Outline

We begin this thesis (Chapter II) with an overview of related work. We first study Markov-based planning algorithms, a popular set of techniques that are often compared to our probabilistic model. Then, we survey current methods for real-time planning and plan execution and present the CIRCA architecture as it existed prior to work reported in this thesis.

In Chapter III, we present the CIRCA-II architecture and describe its components, focusing on how they connect and their complementary roles in overall system functionality. Chapter IV describes in detail the CIRCA-II temporal model, starting with the nondeterministic STRIPS-like transition model used by CIRCA, then describing the compact probabilistic transition-based model used by CIRCA-II, focusing on both the details of the probabilistic model and its use for computing state probabilities and the preemptive task deadlines sent to the scheduler. We present a qualitative comparison of CIRCA-II to the original CIRCA as well as an MDP configured to emulate the functionality of the CIRCA-II planner.

Chapter V contains the methodology by which we detect and react to "unplanned-for" states as they arise. First, a state classification scheme is proposed to assist with the identification of "important" unplanned-for states, which must be detected. We then describe the algorithms used to build tests that can detect these states as they occur during plan execution and propose a methodology for responding to these states that strongly biases the system toward failure-avoidance over goal-achievement when such a tradeoff is required. We discuss how CIRCA-II with unplanned-for states compares to the original CIRCA and also how the addition of the plan cache improves performance, measured in terms of a system's ability to remain safe in its environment.

Chapter VI describes our progress to develop algorithms that enable planner-scheduler negotiation, starting with the single-processor scheduler we have implemented and tested and venturing into algorithms that will also allow expressive feedback to the planner when a multi-resource allocation and scheduling system is implemented in the CIRCA-II architecture. We then describe recent work to introduce fault-tolerance to the CIRCA-II plan-execution platform and discuss methods for adopting Quality-of-Service (QoS) negotiation techniques for allowing both planner and scheduler to degrade system performance, as opposed to requiring that the planner be responsible for all performance degradation computations required to successfully build and schedule each plan.

Chapter VII describes the flight simulation tests we have performed to demonstrate the utility of our algorithms and to explore the use of the CIRCA-II architecture for fully-automating aircraft flight. First, we describe tests with an F-16 simulator that allow us to explore how CIRCA-II can be used to guide the F-16 during fully-automated pattern flight even when specific anomalous situations occur. Next, we describe the next-generation F-16 simulator recently used by Honeywell Technology Center and the University of Michigan for a joint UCAV (Unmanned Combat Aerial Vehicle) demo. We describe the University of Michigan Uninhabited Aerial Vehicle (UAV) project, from hardware to software architecture, and look at the application of CIRCA-II to that vehicle for mission planning and fault recovery tasks.

Chapter VIII summarizes the topics covered and contributions of this thesis and also discusses the diverse set of future research problems that may be addressed in the context of CIRCA-II and more generally the spectrum of hard real-time autonomous systems.

CHAPTER II

RELATED WORK

In this chapter, we describe existing methods for planning and plan execution that are applicable or similar to our work, focusing on their capabilities and limitations with respect to the real-time failure-avoidance properties we require. Figures 2-1 and 2-2 show generic conceptual diagrams of distinct "AI planning" and "real-time" systems, respectively. Traditionally, the AI community has designed architectures that include modules for planning and/or plan execution. The planner takes as input domain knowledge and outputs one or more plans or policies of the type illustrated previously in Figure 1-2. The plan execution system takes plan(s) as input and interprets these plans, ultimately selecting a specific sequence of actions to be executed over time. Environment state feature information (e.g., from sensors) is typically used by the plan execution module to select appropriate actions and/or plans to execute, and may also be provided to the planner to better guide plan development.

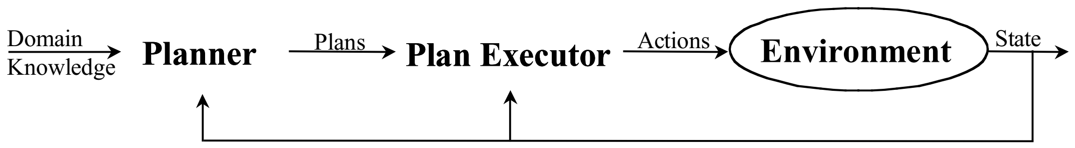


Figure 2-1: A "Generic" Planning/Plan-Execution Architecture.[4]

[4] The 3-T architecture [17] was developed to conceptually describe systems with distinct planning, plan-execution, and reactive layers. In Figure 2-1, we hide any reactive layer in "Environment" because we are not concerned with its operation in this discussion.

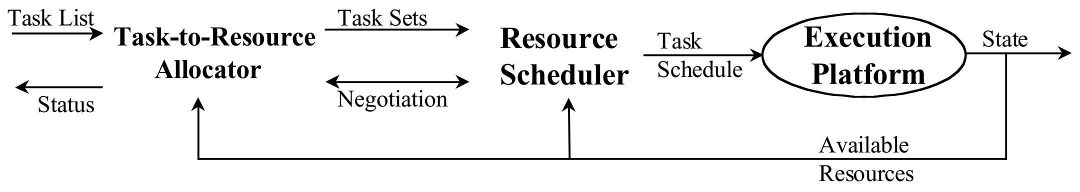


Figure 2-2: A "Generic" Real-time Resource Allocation/Scheduling System.

The real-time community has designed a wide array of algorithms that take as input a set of tasks along with their execution requirements and constraints, allocate resources (e.g., processors, communication channels) for these tasks, then schedule the tasks on each individual resource. As shown in Figure 2-2, the "environment" for a real-time task allocation and scheduling system is the execution platform. Execution resources are typically monitored to provide feedback regarding dynamic changes in the available resource set.

In our work, we combine planning and scheduling algorithms based on techniques developed by planning and real-time systems researchers. To do so, we require a planner with a sufficiently expressive temporal model of the world to succeed in a dynamic real-time environment. Additionally, we require a real-time task scheduler that is sufficiently versatile to accept tasks (actions) produced automatically by a planner as well as provide feedback that is understood by that planner. In this chapter, we describe algorithms and techniques that have been used by others to combine action selection (planning) with the temporal reasoning required for a real-time environment.

In order to use the term "response guarantee" in reference to a system operating in a dynamic environment, we require a planner that is capable of performing a temporal analysis to determine how each action will affect its environment as a function of *when* it is executed in that environment. Actions responsible for maintaining system safety can succeed only if executed prior to a fixed deadline, and thus must execute in *hard real-time*. The concept of a hard real-time system is illustrated in Figure 2-3, with the compromise in safety after a hard deadline passes represented by a sharp drop in reward. This behavior contrasts with *best-effort* or *soft real-time* tasks in which reward (or performance) degrades more gracefully over time. As we will describe below, many of the popular "real-time" planning and/or plan execution architectures exhibit only best-effort execution capabilities. Thus, they can be at best *coincidentally* real-time when applied to a hard real-time domain.[5]

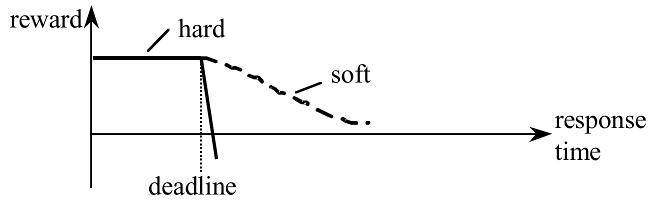


Figure 2-3: "Hard" versus "Soft" [Best-effort] Real-time System.

This chapter begins with a discussion of related planning research. Because we admit systems with limited resources and imprecisely-specified domain knowledge, we utilize a probabilistic planning model for our research. The Markov Decision Process (MDP) [11],[45] is perhaps the most flexible existing probabilistic planning model, thus we devote a section of this chapter to a discussion of this popular technique. Next, we describe research in real-time planning in which researchers explicitly place bounds on deliberation time to allow response before a hard task deadline occurs.

Because state-space planning is generally NP-complete, such planning algorithms are generally not viewed as sufficient for a stand-alone hard real-time system (e.g., without any supporting scheduling and/or plan execution processes). As a result, a

[5] Some researchers argue that such systems can be used for hard real-time domains by performing an extensive battery of tests that verify real-time performance. We agree with this verification method in theory, but in practice one must be able to demonstrate that all possible situations have been encountered during testing. In our opinion, this diminishes the attractiveness of the system and provides a formidable challenge for each application of the architecture to a new problem.

multitude of architectures have been developed that incorporate distinct plan-execution components to augment (or replace) a planner. This design has improved the AI community's ability to build systems capable of automating complex and dynamic domains, as demonstrated by their success in real-world applications. However, many of these architectures exhibit the coincidental real-time response for which we are skeptical in safety-critical systems. We conclude this chapter with a description of the Cooperative Intelligent Real-time Control Architecture (CIRCA) [51],[53] which was explicitly designed to guarantee failure-avoidance via integrated planning/scheduling/plan-execution software. We have built upon the CIRCA architecture in this dissertation, thus we also discuss its limitations and preview how we address these issues in CIRCA-II.

Probabilistic Planning: The Markov Decision Process

The Markov Decision Process (MDP) model is the basis for many state-of-the-art probabilistic planning (policy construction) algorithms, and is very attractive due to its ability to convert state transition probabilities into optimal plans, where optimality is measured in terms of a value or utility function. Described in [11], the general MDP is given by $M = (S, A, P, R)$. In this representation, S is a finite set of N_S states where N_S represents the combinatorial set of all state features and their possible values, A is a finite set of N_A actions, P is a state-transition matrix, and R is a reward function. The MDP presumes the system evolves in *stages,* where the occurrence of some event (or action) results in the transition from some state t to the next state $t+1$. Although the progression of stages need not necessarily correspond to a progression in time, this is a valid mapping appropriate for our "real-time plan" development purposes.

Several terms are used to classify systems within an MDP framework. The *Markov assumption* [11] says that "knowledge of the present state renders information about the past irrelevant to making predictions about the future". This is a required property of an MDP system to allow a tractable representation of state transition probabilities. Also, in any system that can presume the effects of each event depend only on the current state, and not the stage (i.e., time) at which the event occurs, we say the MDP is *stationary* and can be represented using only two stages. Otherwise, for a *T-stage nonstationary Markov chain*, state transition probabilities are dependent on the current stage number (e.g., amount of time that has passed). Due to this dependence, in the worst case (fully-connected state-space), T transition matrices of size $N_A x N_S x N_S$ must be provided. Using the notation in [11] and [45], the probability of transitioning from a state s_i to state s_j at stage t is then given by p^t_{ij}, and the full state transition matrix P is of size $Tx\ N_A x N_S x N_S$.

One key property of the basic MDP is that all state features must be observable in each stage so that the correct action for each state can reliably be chosen. A *partially-observable MDP* (POMDP) [13] is defined as an MDP in which some state feature measurements are either noisy or unavailable, in which case probabilistic distributions regarding the likelihood of unobservable state features must be incorporated into the model instead of directly observed. The requirement to use a POMDP for system modeling dramatically increases planning computational complexity, and thus is generally avoided when it is possible to measure all state feature values.

Our emphasis in this research is on the development and execution of "control plans" (defined in Chapter 1) in which all reactions are sufficiently accurate and timely to guarantee safety in dangerous dynamic environments. We presume that a system can measure all state features, although in some cases with non-negligible cost, thus we do not require a POMDP formulation for our problem. However, we do require a system capable of reasoning about the *time* at which each event will occur so that the system can explicitly compute all action deadlines required to preempt states that represent catastrophic failure. Further, in MDP terminology, we require a state transition matrix that accurately reflects the effects of selecting different action execution deadlines so that we can construct an optimal (or at least sufficient) control plan in which all failure-avoidance actions will be guaranteed to execute in hard real-time on the limited set of plan execution resources.

To account for the probabilistic effects of varying action execution deadlines, an MDP must be augmented such that one action is specified for each deadline available for each action. So, for example, suppose an automated aircraft system has an action "emergency-land". Then, the MDP transition matrix will require actions such as "emergency-land-in-1-minute", "emergency-land-in-2-minutes", etc., since the probabilities of other transitions (e.g., crash) will be conditional on the amount of time that passes before the [failure-avoidance] activity completes.[6] This addition to the MDP effectively increases the set of actions from the previous N_A to $N_A \times N_D$, where N_D represents the number of unique deadlines that may be assigned to each action.

We seek a general technique for reasoning about the temporal characteristics of exogenous events and actions during planning. Figure 2-4 shows a simple three-stage state transition diagram excerpt illustrating an "engine-failure" exogenous event coupled with actions (deadline excluded from figure labels) to start the aircraft engine, fly to new locations, and "emergency-land" if the engine fails during flight. Now, presume the system begins at stage *t* with two possible states, one in which the engine has been running since some previous time and the other in which the engine is off. Now, as will be discussed in Chapter 4, a state transition such as "engine-failure" can have very different probabilistic properties depending on factors such as how long the engine has been operating, particularly as the engine approaches or passes the end of its normal

[6] For actions that execute with different deadlines, either the time between stages in an MDP must vary in accordance with the action execution deadline or else the action must continue executing across multiple stages. We presume the former because the latter requires a violation of the Markov assumption unless special features are added to the state to effectively "remember" which action is executing and when it began, further increasing MDP complexity.

operating life.[7] So, presuming that the time between stage *t* and *t+1* is non-trivial, the likelihood of an engine-failure from the "Engine=ON" state in stage *t+1* may be dependent on the path in which the system arrived at stage *t+1*. Thus, the Figure 2-4 model violates the Markov assumption.

As discussed in [11], any non-Markovian model whose dynamics (e.g., event probabilities) depend on at most *k* previous stages can be converted to a larger Markov model.[8] This conversion to "state form" [46] requires that information be added to each state to keep track of how much time has passed (e.g., since the engine has been started) so that state transition probabilities can accurately reflect these effects. Note that as *k* increases, the number of states in the "converted Markov model" also increases, in the worst-case exponentially in *k* (i.e., to N_S^k). This worst-case represents the fully-connected situation in which all possible states are expanded in all stages and all state information must be "remembered" from each of the previous *k* stages.

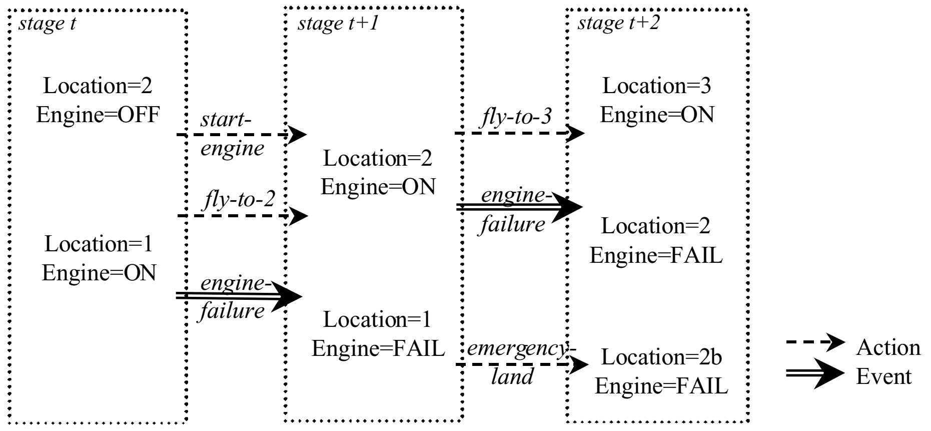


Figure 2-4: Three-Stage State Transition Diagram.

[7] Note the effects are exaggerated here for effect, so the reader should not allow this dissertation to instill a fear of flight, which is statistically much safer than travel by car.

[8] As limiting cases, *k=1* represents a "naturally" Markovian model and *k=T* (the maximum number of stages expanded) indicates that all stages must be "remembered".

For small k and large number of stages T (i.e., translatable for our purposes to time horizon $t_{horizon}$), the above "k-level" (i.e., k-stage memory) conversion process is perhaps the most efficient method for utilizing a generic MDP planner. In the special case where k is equal to the number of stages and the model is Markovian so long as each state "remembers" the current stage number (corresponding to time in our model), a system can utilize the standard non-stationary MDP described above. For the Figure 2-4 example, knowledge of stage number is *not* sufficient because engine operation time varies between states within a particular stage. Thus, we adopt the "k-level" conversion model that includes actions for each possible deadline as the MDP-equivalent for "control plan" development as defined in Chapter 1. As discussed above, this MDP model has a transition matrix P of worst-case size $(N_A\ xN_D)$x N_S^k x N_S^k.

It is not difficult to observe that this MDP model grows large very quickly. Methods such as Bayesian Networks [61] have been developed to efficiently represent conditional state probabilities like those found in P. Then, algorithms such as Policy Iteration and Value Iteration [11] are used to compute an optimal policy (i.e., mapping of actions, along with deadlines for our model, to states). For large problems, recent advances such as factoring the state-space [14] and using algebraic decision diagrams to solve factored MDPs [30] have been developed. We have not yet verified whether these methods will be capable of significantly reducing complexity for our problems of interest.

We have devoted this section exclusively to a discussion of MDPs, but have opted for a probabilistic state-space planner with a STRIPS-like [18] state transition representation for our CIRCA-II architecture. This decision was based on a number of factors. First, since computing flexible action deadlines to accommodate execution on limited resources is a key part of each control plan, we find it overly restrictive to require a knowledge base in which a unique action must be specified for each possible deadline. Additionally, we have observed that the transitions and "intuitive" features associated with hard real-time domains often require multi-level histories as was illustrated by Figure 2-4, and that the development of the "k-level" model for large k will be a difficult undertaking in itself. For these reasons, we present an approximate probabilistic state-space planning model in this dissertation, sacrificing MDP optimality for knowledge and state-space compactness while maintaining the ability to select actions and corresponding deadlines that guarantee hard real-time failure-avoidance during plan execution.

Real-time Planning

In this section, we briefly discuss methods to restrict planning time so that real-time response deadlines are met. Such systems typically employ approximate planning techniques, and thus have the advantage of controllable and predictable real-time execution. However, due to the unavoidable complexity of planning, real-time planners often have difficulty in guaranteeing result accuracy simultaneously with timeliness, as discussed in [51]. Nevertheless, powerful techniques have been developed for real-time planning, and in the future we hope to incorporate an algorithm to bound planning time that will most likely be based on work described in this section.

One of the most basic approaches to time-limited planning is to build a minimal plan then iteratively improve this plan until planning time expires. This idea has been labeled *anytime planning* [15], and has been implemented in many interesting algorithms. In an anytime system, a planner must produce a very approximate plan almost immediately; this plan may be a random guess in the worst case. As time passes, the plan is refined to become more accurate, until the planner is interrupted by the anytime monitoring process for the best result it has computed given planning time available so far. Abstraction planning algorithms such as that in [10] and [31] illustrate the utility of anytime planning. For a variety of problems, the combined anytime-abstraction approach is sufficiently fast to produce high-quality approximate plans. Additionally, different anytime algorithms can be merged to produce an optimal result given deliberation time constraints [74],[75]. However, even the most efficient anytime algorithm must assume that sufficient time is available for a “minimally-accurate plan” to be developed.[9]

Design-to-time planning [21] presumes the system can compute or approximate in advance the amount of time that will be available for planning. Then, the planning process is tailored via the choice of parameters (e.g., level of abstraction, planning technique) so that a solution will be returned within the available time. This approach presumes there is a function that allows planning time to be predicted based on the choice of parameters, a difficult computation for a number of NP-complete planning problems in which worst-case execution time (based on domain knowledge size) will often be much larger than available computation time, although planning average-case execution time may be available. A major advantage of design-to-time planning is that the system will know in advance whether it can expect to succeed given the available planning time. So, in systems that begin in a “safe” environment, if the planner predicts it cannot produce minimally-accurate plans in real-time, it can remain safe by never venturing away from safety (e.g., an aircraft will never leave the ground). Alternatively, if the planner predicts it can produce minimally-accurate plans for the fastest responses required for that domain, the system may even be able to make guarantees regarding system safety.

Design-to-criteria scheduling [70], based on the earlier design-to-time research, is the soft real-time process of finding an execution path through a hierarchical task network so that all execution constraints are met, including real-time deadlines, cost limits, and plan quality requirements. This procedure incorporates a probabilistic model to make approximations when required, as does the planner we describe in this work. However, discontinuities in the utility functions used to guide the design-to-criteria

[9] For the safety-critical systems considered in this dissertation, a minimally-accurate plan must be capable of responding in a timely manner to all events that lead to system failure.

processes may compromise the result, as would be the case when a hard real-time response deadline is exceeded.

In this dissertation, we do not directly address the problem of real-time planning due to the tradeoff between planner response timeliness and plan accuracy. However, as discussed in Chapter VIII, we will be revisiting the real-time planning issue in follow-on research as we work to make dynamic replanning for anomalous world states *temporally-bounded*[10] when required.

Real-time Plan Execution

When deadlines are so tight that none of the available real-time planning techniques provide sufficiently accurate results, they must either be integrated with or replaced by real-time plan-execution algorithms. At the opposite end of the "planning - plan execution" spectrum, consider the concept of a Universal Plan set [63]. The existence of such a set is desirable because it completely avoids dynamic planning by including responses for all modelable states, regardless of whether they are reachable given the initial state and goals for a particular mission (e.g., flight).[11] However, as discussed in [23], for complex domains, bounded planning resources (e.g., memory, disk storage capacity) may make creation of the complete Universal Plan set infeasible, and hard real-time plan execution constraints may prohibit response guarantees due to the time required for retrieving the appropriate plan(s) from such a large database.

Many researchers have addressed the challenges associated with utilizing a predefined plan set. Because such plan databases can be large, if not "Universal", plan-

[10] We use the term *temporally-bounded* to indicate that a process is timely but not necessarily accurate, both of which are required for a designation of hard real-time.

[11] The Universal Plan concept is distinct from the MDP policy in that the MDP only builds into its policies reactions for states that are reachable with *non-zero* probability from the known initial state(s).

execution architectures employ a wide array of efficient plan retrieval procedures that allow successful execution in real-time environments. First, the RAPs architecture [19] specifically addresses the issue of fast reactions in complex domains. Using a set of production-like rules, the system takes sensor information about its environment and selects appropriate reactions using a set of predefined rules (or RAPs). Similarly, the Procedural Reasoning System (PRS) [33] addresses real-time plan execution by employing a plan (or procedure) database, and reacting to situations as they occur by retrieving one or more of these plans for execution.

Searching through a procedure or rule database has been shown to be sufficiently fast for a variety of real-time domains, including numerous mobile robot experiments such as those in [32]. However, absolute real-time guarantees are impossible to achieve in RAPs and PRS unless they are augmented with a module to prove that the worst-case database search-and-act time is less than or equal to the fastest response that might be required for failure avoidance, as was discussed for C-PRS in [32]. Additionally, even with an execution time proof, in many cases the absolute worst-case search-and-act time would be greater than the minimum response time required so the proof would only show that the system could fail in the worst case.

Architectures such as CYPRESS [71] and SOAR [40] have demonstrated the ability to succeed in real-time environments, including SOAR's success during a variety of military simulation exercises [34] and the world of RoboCup [69]. Both architectures combine efficiency with flexibility by using a reactive plan-execution system whenever a response is available and by performing dynamic planning (subgoaling) otherwise. However, neither explicitly reasons about task deadlines or worst-case resource utilization, so they fall under the classification of "coincidentally" real-time systems defined previously.

A number of researchers have begun to carefully consider hard real-time constraints when applying planning and plan-execution techniques to real-world problems. For example, in [27], a set of conditional schedules have been demonstrated to provide predictable plan-execution response times for an aircraft avionics problem. Such systems address the same critical hard real-time issues we consider in this dissertation. However, our work is distinguished by specifically separating the failure avoidance problem from goal achievement, and by considering classes of both probable and improbable situations that require hard real-time responses.

The New Millennium Remote Agent (NMRA) architecture used for the Deep Space One (DS-1) spacecraft [55] employs a constraint-based planner and scheduler to allow real-time responses to a variety of situations. Although NMRA may be generalized to other domains, it relies on the fact that DS-1 has brief periods requiring hard real-time response (e.g., a fuel burn sequence during orbit insertion) interspersed with longer time periods in which significant "slack time" is available for planning/scheduling.[12] A basic premise in our work is that we cannot rely on the occurrence of such slack intervals, thus we need to be able to recall pre-built contingency plans to guarantee safety in time-critical situations, even if these reflex actions actually make the system diverge from its task-level goals. With such an assumption, our system is not as appropriate as NMRA for a spacecraft like DS-1, where achieving major task-level goals is as time-critical and crucial as maintaining spacecraft safety. Instead, we focus on problems in which a clear distinction can be made between failure-avoidance tasks and less-important goal-achievement tasks, but allow our system to operate continuously in a "dangerous" dynamic environment, as illustrated in Chapter VII for the autonomous flight domain.

[12] This limitation is necessitated by the limited resource set (i.e., one processor) on the DS-1 spacecraft that prohibits any appreciable planning or scheduling simultaneously with other critical low-level activities.

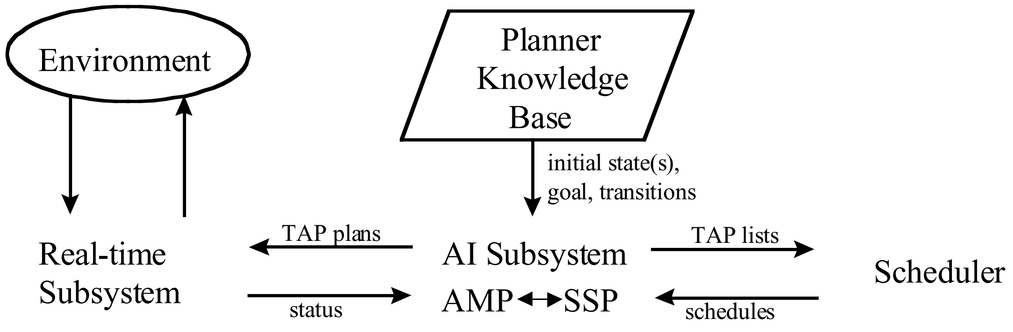


Figure 2-5: The Cooperative Intelligent Real-time Control Architecture.

CIRCA: Reasoning About Real-time Plan Execution

The Cooperative Intelligent Real-time Control Architecture (CIRCA) [51],[53] combines aspects from planning and plan-execution systems, and was designed to explicitly separate the time-consuming planning process from real-time plan execution. Using this approach, the planner reasons *about* real-time, but does not have to make approximations that permit planning *in* real-time. To build real-time plans, CIRCA's planner employs a time-dependent state transition model and a representation for system "failure" such that plans contain two classes of *test-action pairs* (TAPs): "guaranteed" with hard deadlines for failure avoidance, and "soft real-time" with best-effort execution for goal achievement.

As shown in Figure 2-5, CIRCA is divided into an AI subsystem that includes a high-level automated mission planner (AMP) and state-space planner (SSP), a real-time task scheduler, and a real-time [plan execution] subsystem (RTS). The AI subsystem is responsible for compiling a set of actions to achieve its overall mission goals while guaranteeing failure avoidance. The guaranteed set of these TAPs is sent to a real-time scheduler, and if scheduling is successful, downloaded to the RTS for execution. If scheduling is not successful, a simple "fail" status is returned from the scheduler. The state-space planner then backtracks to (hopefully) find a different set of actions that will still be guaranteed to avoid failure while achieving the same goal. Otherwise, the AMP must somehow modify the planning problem so that a schedulable plan may be developed.[13]

Because of its careful consideration of real-time failure avoidance, we have adopted CIRCA as the basis for the architecture presented in this dissertation, aptly titled CIRCA-II. The original CIRCA has many desirable properties, including its real-time execution guarantees and its separation of planning from plan execution such that significant (and often unpredictable) planning approximations are not required for real-time performance. However, CIRCA also makes significant assumptions which may cause difficulties in complex real-time environments. First, CIRCA presumes that any executing plan can maintain safety *indefinitely*. This assumption is valid for many domains, such as a mobile robot that can either stop (in a benign environment) or perform a fixed safety-maintenance loop (in a closed, predictable world) forever. However, other domains cannot include “stop” or indefinite “safety-maintenance” actions. For example, during fully-automated flight, the aircraft cannot remain safe indefinitely once it has left the ground, since fuel must continuously burn and a variety of anomalous situations may prohibit the aircraft from reaching its planned-for landing destination.

Another important CIRCA assumption is that there exists at least one *schedulable* plan that can achieve the complete set of task-level goals while guaranteeing failure avoidance. Because of CIRCA’s non-deterministic state-space model, it is impossible to make any approximations (e.g., remove improbable states), thus the planner would simply fail if no one schedulable plan could be produced after exhaustive backtracking. In this dissertation, we describe algorithms that directly address these limitations and thus

[13] In the original CIRCA implementation, the AMP relied solely on human intervention to manually redefine the planning problem (e.g., by removing knowledge of some failure modes from the model) when SSP backtracking alone could not provide a schedulable plan. Theoretically, the AMP could have automatically redefined mission goals or somehow removed state transitions from the SSP knowledge base; parallel research efforts at Honeywell Technology Center are focusing on better defining such procedures.

distinguish CIRCA-II from CIRCA. Topics include inclusion of a probabilistic planning model that allow planning approximations to enhance schedulability (Chapter 4), description of a methodology by which CIRCA-II detects and handles "unplanned-for" states, thereby accounting for [limited] domain knowledge imprecisions (Chapter 5), and techniques utilized to provide more expressive scheduler-to-planner feedback to guide backtracking operations prompted by a plan scheduling failure (Chapter 6).

CHAPTER III
CIRCA-II ARCHITECTURE

The original CIRCA architecture was designed to provide guarantees about system performance with limited resources, given closed-world assumptions about transition model accuracy, preemptive action schedulability, and the ability of the system to maintain safety indefinitely during plan execution. CIRCA-II is based on CIRCA, but is distinct in that its state-space planner is based on a probabilistic rather than non-deterministic state-space model (see Chapter IV), admits incomplete knowledge and graceful performance degradation via detection and handling of "unplanned-for" states (see Chapter V), and extends the communication protocol between planner and scheduler to help guide planner backtracking toward a schedulable plan (see Chapter VI). In this chapter, we describe the CIRCA-II architecture and its components, all of which will be referenced throughout this dissertation. We focus on module interconnections in CIRCA-II, and provide a high-level description of component functionality. See Appendix A for a more detailed description of the C++ CIRCA-II software implementation, which focuses on custom architectural components in terms of their current implementation as well as possibilities for future enhancements.

Figure 3-1 illustrates the CIRCA-II architecture. At the highest level, the architecture is divided into a Planning Subsystem and a Plan-Execution Subsystem. CIRCA and CIRCA-II both draw distinct boundaries between real-time and non-real-time processes to facilitate reasoning about the hard real-time guarantees required for failure-avoidance during plan execution. In CIRCA-II, we have designated that the Planning Subsystem include all processes for which we cannot easily define reasonable worst-case execution properties. At best, CIRCA-II will be able to complete Planning Subsystem operations in *coincidental real-time*. Thus, we relegate the majority of planning and scheduling operations to occur offline before the system ever enters its "dangerous" environment. Conversely, safety-critical tasks in the "Plan-Execution Subsystem" require hard real-time execution. This module includes a "Plan Executor" with analogous functionality to the original CIRCA Real-Time Subsystem (RTS) [53], and a new "Plan Dispatcher" that is responsible for managing a Plan Cache, communicating with the Planning Subsystem, and starting/killing all individual Plan Executor processes.[14]

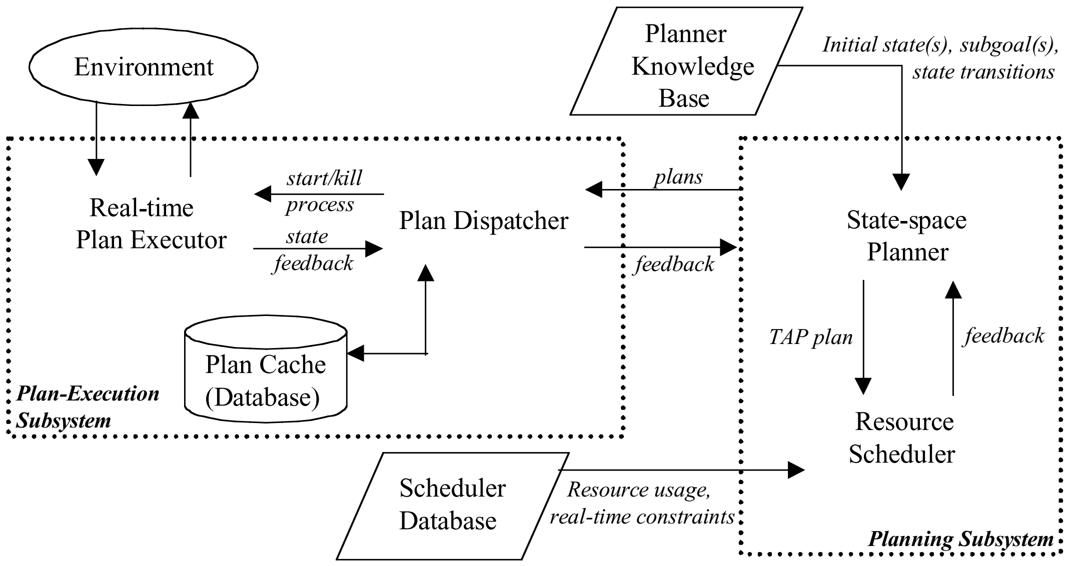


Figure 3-1: CIRCA-II Architecture.

[14] In CIRCA-II, each new plan is executed as a separate process. This is markedly different from the original CIRCA RTS, in which a set of plans (including multiple goal-achievement plans but no contingency plans) was downloaded from the planner and stored in a buffer until executed within the single RTS process. We made this change to move CIRCA-II's plan executor toward a real-time-thread execution model and expect further change as we move toward a multi-resource execution platform as well.

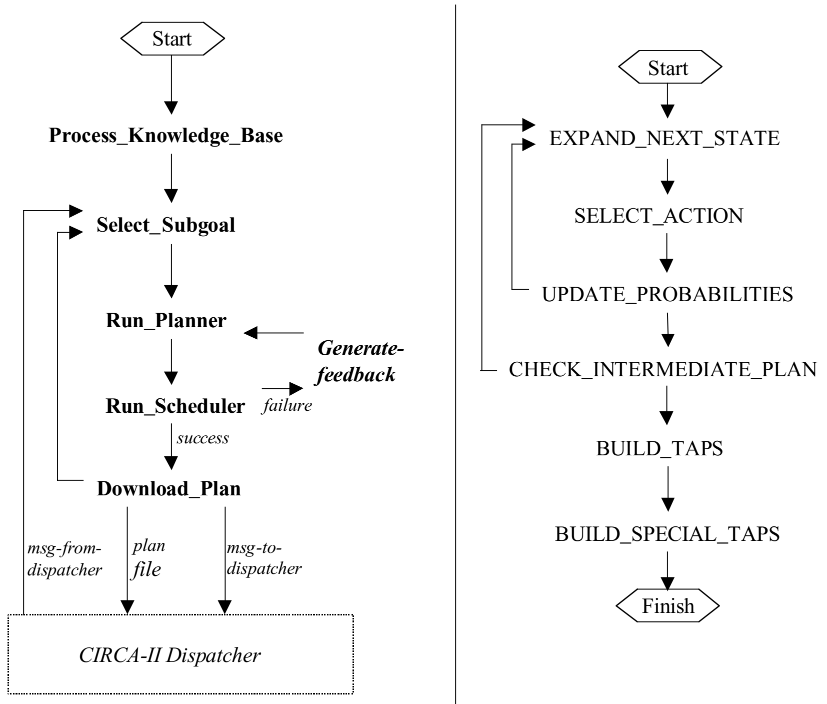
Upon startup, CIRCA-II builds and schedules the set of "nominal" plans required to achieve its pre-defined mission goals and a set of "contingency" plans specifically developed to be retrieved in hard real-time for failure-avoidance purposes (see Chapter V for more details on nominal and contingency plans). Both plan sets are developed offline and stored in the plan cache on the hard real-time Plan-Execution Subsystem. Next, the planner signals that plan execution should begin, and the first goal-achievement-oriented *nominal plan* stored in the cache begins execution. If only likely, planned-for events occur, the sequence of nominal plans will execute to completion. However, if a transition leads out of the “planned-for” state set in any goal-achievement plan, the plan cache module reacts with a pre-computed failure-avoidance *contingency plan* to maintain safety (if necessary), then the planner develops a new set of actions to redirect the system toward its goals. In the following sections, we briefly look at the functionality inside the Planning and Plan-Execution Subsystem modules. Again, for further details regarding specifics of the implementation, see Appendix A.

Planning Subsystem Operation

Figure 3-2a and 3-2b show the algorithm used by the Planning Subsystem to create a plan that executes with hard real-time guarantees of failure-avoidance. Upon startup, the planner processes initial states and transitions (Process_Knowledge_Base in Figure 3-2a), then selects a subgoal from the knowledge base.[15] Next, the planning process is initiated, as will be described below. The state-space planner produces as output a set of TAPs (test-action pairs) which is then passed into the real-time scheduler. This resource scheduler, algorithms for which are discussed in Chapter VI, builds a cyclic

[15] A pre-defined sequence of subgoals to achieve are currently specified in the knowledge base. Ideally, subgoals would be automatically created from a single high-level goal. Researchers at Honeywell and the University of Michigan [4] are in the process of designing methods to automatically generate the goals to be achieved within each plan.

schedule for all guaranteed TAPs assuming worst-case execution properties, as in CIRCA [51]. If scheduling is successful (i.e., all task deadlines can be met given the available plan-execution resources), the plan is downloaded to the Plan Dispatcher in two pieces: a plan file to be compiled into a plan execution process and a message containing a decision-tree [57] database index specifying the set of states for which this plan should execute. Otherwise, the plan must be modified so planning is again initiated, with scheduler feedback (see Chapter VI) used to guide backtracking.



a) CIRCA-II Planning Subsystem Procedure. *b) "Run_Planner" Procedure.*

Figure 3-2: CIRCA-II Planning Subsystem.

A loop from the Download_Plan operation to Select_Subgoal is shown in Figure 3-2a because CIRCA-II continues building nominal plans for each subgoal in the specified sequence, as well as contingency plans for failure-avoidance should unplanned-for states be reached. Only when this set of plans is fully developed does the dispatcher direct the system to enter its dangerous environment via execution of the first nominal plan. During plan-execution, the CIRCA-II Dispatcher may send a "replan" message to the planner to accommodate "safe" unplanned-for states. The planner uses state feature feedback information to select a new subgoal that can be achieved from this state (by matching user-specified subgoal "preconditions" from the knowledge base), then builds and downloads the new plan to the dispatcher.

As in CIRCA, CIRCA-II adopts a forward-chaining planner to expand the state-space from initial to goal state(s). The probabilistic planner enables the use of a best-first search state expansion strategy with states ordered from most probable to least probable. Although this expansion ordering is not crucial in the current CIRCA-II implementation, future research to implement real-time planning (see Appendix D) with an anytime component in CIRCA-II may rely on best-first search ordering to maximize plan quality. As shown in Figure 3-2b, for each "reachable" state, the planner expands the state (i.e., identifies matching transitions and the "child" states resulting from those transitions), selects an action for that state (if any are judged as beneficial either for failure-avoidance or goal-achievement), and then updates the probabilities to include effects of this newly-expanded state. As will be described more completely in Chapter IV, during probability computations, any execution constraints for actions requiring hard real-time execution will also be updated/added so that the probability of system failure (i.e., traversing a temporal transition to failure *(ttf)*) remains below a small threshold P_{thresh} from each state.

The state expansion cycle terminates with a comprehensive check of the "intermediate plan" (i.e., action and associated deadline for each reachable state), which verifies that the plan can achieve the specified subgoal.[16] If no goal state is in the reachable set, the planner backtracks to select different actions until at least one goal path is identified. After a valid intermediate plan is constructed, the set of planned actions is

[16] The plan need not be re-checked for failure-avoidance because Update_Probabilities verifies preemption of all *ttfs* during plan construction.

compiled into *Test-Action Pairs (TAPs)*, one TAP per unique planned action, along with *special TAPs* constructed to report the detection of *unhandled states* (i.e., states that were not explicitly handled during nominal plan development) as well as to flag when *goal states* (i.e., state with features matching all specified subgoal features) are reached.

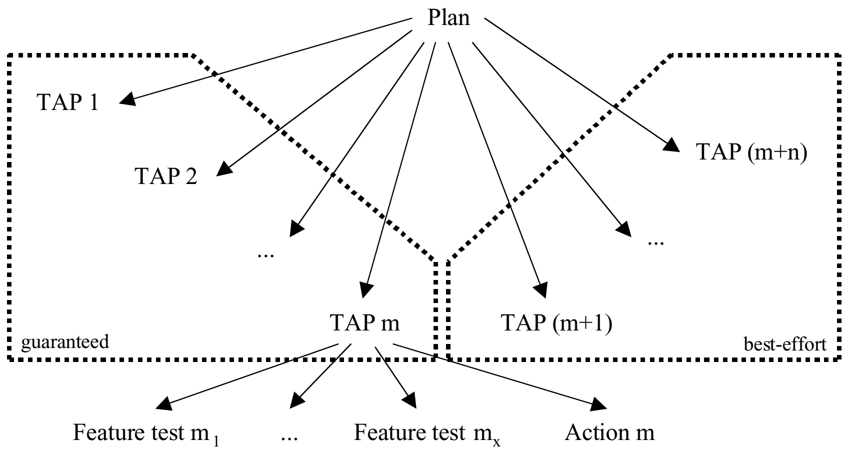


Figure 3-3: CIRCA Test-Action Pair (TAP) Plan Composition.

Figure 3-3 illustrates the structure of a plan that will be sent to the scheduler then downloaded to the dispatcher. The set of planned (and special) TAPs is divided into two classes: *guaranteed* (hard real-time) and *if-time* (best-effort). Guaranteed TAPs must be inserted into a schedule that meet their associated *ttf* preemption deadlines, while best-effort TAPs are executed during slack time as part of an *if-time server* TAP (see Appendix A). Each TAP contains one action but may execute in multiple states (e.g., all reachable states for which the TAP action has been selected). CIRCA-II (like CIRCA) uses ID3 [57] to build a set of feature tests which will return a true/false value indicating whether the current state requires execution of the TAP action. As depicted in Figure 3-3, these feature tests and action together comprise a TAP, and are critical to consider during scheduling (e.g., worst-case TAP execution time is the sum of all feature test and action worst-case execution times) and subsequently during plan-execution (e.g., since the system must perform *Action m* only when feature tests m_1-m_x return values that match those in the associated TAP test).

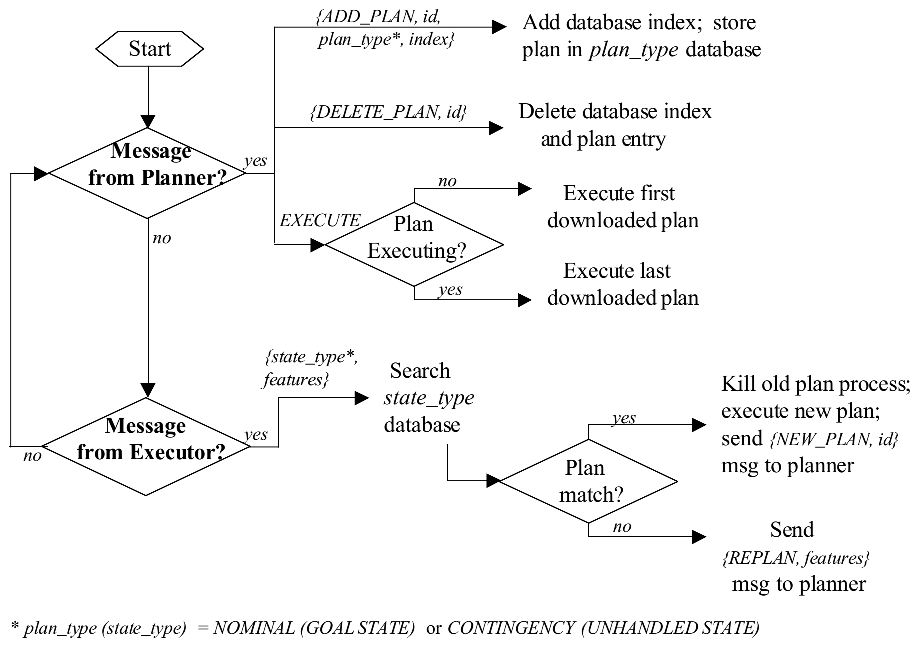

Start
Message from Planner?
yes
no
{ADD_PLAN, id, plan_type*, index}
Add database index; store plan in plan_type database
{DELETE_PLAN, id}
Delete database index and plan entry
EXECUTE
Plan Executing?
no
Execute first downloaded plan
yes
Execute last downloaded plan
Message from Executor?
yes
no
{state_type*, features}
Search state_type database
Plan match?
yes
Kill old plan process; execute new plan; send {NEW_PLAN, id} msg to planner
no
Send {REPLAN, features} msg to planner
* plan_type (state_type) = NOMINAL (GOAL STATE) or CONTINGENCY (UNHANDLED STATE)

Figure 3-4: CIRCA-II Plan Dispatcher.

Plan-Execution Subsystem Operation

The CIRCA-II plan execution system is responsible for executing the TAP plans developed and scheduled by the Planning Subsystem in hard real-time when required for failure-avoidance. As shown in Figure 3-1, the Plan-Execution Subsystem contains the Plan Dispatcher (including the plan cache) and the Plan Executor. Upon CIRCA-II startup, we presume a system is in an indefinitely safe state (e.g., a commercial aircraft is parked at an airport gate) since we may require extensive [unbounded] planning and scheduling operations to populate the plan cache sufficiently for safe entry into the "dangerous" environment (e.g., aircraft engine spool-up then takeoff from an active runway). In this section, we first discuss operation of the new CIRCA-II Plan Dispatcher. Next, we discuss features internal to the Plan Executor that allow real-time execution guarantees both while executing a single plan and while switching to a new plan from the CIRCA-II plan cache.

Figure 3-4 illustrates the functionality of the CIRCA-II Plan Dispatcher. This program module is labeled "dispatcher" because its main program is simply comprised of a loop that waits for and processes messages from the CIRCA-II Planning Subsystem and Plan Executor, similar to the sequencing layer in the ATLANTIS architecture [17]. However, functions triggered by messages are responsible for all plan database operations, plan-execution process management, and state feedback processing. Ultimately, dispatcher operation is dictated by control commands downloaded by the planner or inserted into plans to be triggered during execution (e.g., unhandled state feedback). Upon dispatcher startup, the plan cache is empty and no plan is executing. Thus, the dispatcher sits idle until the planner begins downloading the initial set of plans to the cache. The planner notifies the dispatcher of each new plan with an "ADD_PLAN" message that includes a plan ID (for common plan reference between dispatcher and planner), plan type (nominal or contingency), and ID3-generated decision tree index to be used by the dispatcher to match this cached plan to state feedback. Alternatively, the planner may send a "DELETE_PLAN" message to prune an entry from the plan cache when the planner determines it is no longer required.[17]

After the planner fully populates the plan cache with the set of nominal and contingency plans it expects to require for this execution sequence, it downloads an "EXECUTE" message to the dispatcher. The dispatcher then executes (i.e., starts the plan-execution process associated with) the first [nominal] plan downloaded,

[17] This function is implemented and tested in the dispatcher, but the decision-making mechanisms required to select plans for deletion have not yet been incorporated into the Planning Subsystem. We suspect such software will be incorporated concurrently with algorithms to determine when the cache should "remember" frequently-used plans between CIRCA-II runs.

corresponding to the first subgoal to be achieved. At this point, both planner and dispatcher monitor the system for incoming messages until the Plan Executor feeds back state information to the dispatcher, corresponding to either a "GOAL" or "UNHANDLED" state, either of which is detected via one of the executing plan's special TAPs. The dispatcher will then search the appropriate cache partition (i.e., nominal or contingency) for a matching plan using the supplied state feature feedback. If a match is found, the dispatcher kills the executing plan process, starts the new plan process, and sends a "NEW_PLAN" message with the new plan ID to the planner.[18] Otherwise, the dispatcher sends the state feature information back to the planner as part of a "REPLAN" message. Then, after the Planning Subsystem downloads the plan developed as a result of the "REPLAN" message, it again transmits an "EXECUTE" message indicating that the dispatcher should pull the most recent plan from the cache and execute it.

As described above, plan execution processes are spawned and killed by the CIRCA-II Plan Dispatcher. However, they are responsible for ultimately carrying out all planned actions and remain the only link between CIRCA-II and its environment, as shown in Figure 3-1. Figure 3-5 outlines the functionality of each CIRCA-II Plan Executor process. Upon startup, the TAPs and TAP schedule for that plan are initialized. Then, the main program loops over the cyclic schedule of guaranteed TAPs.[19] If a guaranteed TAP takes less than its worst-case execution time, the quantity used for scheduling, then *slack time* will be available before the next scheduled TAP must begin.

---

[18] The worst-case execution time (wcet) for each plan switch must include the worst-case plan cache search time plus delays due to killing the old plan process, then starting and initializing the new plan process. To-date, our test domains have had TAPs with wcets on the order of 0.1+ seconds, and plan switch overhead has been comparatively insignificant thus easily included. We will be forced to more carefully consider plan switch overhead for a domain with TAP wcets falling into the millisecond range or less.

[19] The guaranteed TAP schedule may include the *if-time server* if plan-execution resources were under-utilized even after all failure-avoidance TAPs are inserted.

During this interval (if it exists), the Plan Executor calls the *if-time server* to run best-effort (if-time) TAPs. Appendix A discusses the current if-time server options implemented within the CIRCA-II Plan Executor that build upon the "Round-Robin" procedure utilized by the original CIRCA [51].

In the chapters that follow, we will examine in more detail the major research advancements implemented within CIRCA-II. First, we will describe the probabilistic model built into the CIRCA-II planner. Next, we will look from a more general perspective at how CIRCA-II can detect and handle "unplanned-for" states, and describe our rationale and procedures for building the nominal and contingency plans that populate the CIRCA-II plan cache. We will then focus on a very specific part of CIRCA-II: the interface between the planner and scheduler within the Planning Subsystem, and finally describe how CIRCA-II has and is continuing to fit into a system for safe, fully-automated aircraft flight.

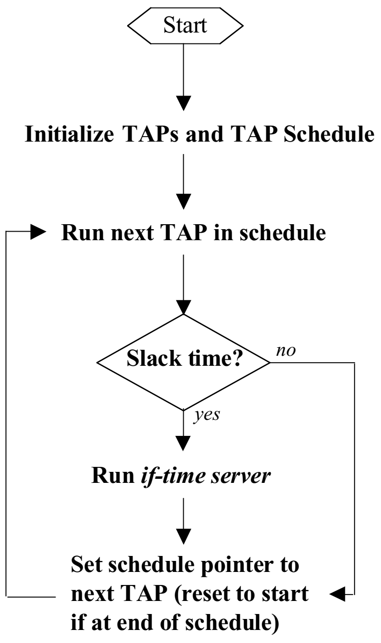

Start
Initialize TAPs and TAP Schedule
Run next TAP in schedule
Slack time?
no
yes
Run if-time server
Set schedule pointer to next TAP (reset to start if at end of schedule)

Figure 3-5: CIRCA-II Plan Executor.

CHAPTER IV
THE PROBABILISTIC TEMPORAL MODEL IN CIRCA-II

Although many probabilistic planners have been developed, most do not consider event probabilities as functions of time. Many real-world events may be accurately represented only with such time-dependent probability functions, and a reliable system must also react with sufficient speed and accuracy to preempt all dangerous events, effectively forcing the probabilities of those events close to zero for all times. In CIRCA-II, we employ an algorithm that prioritizes states of the world by the probability of visiting each at least once, which we define as a state's probability. We perform this computation so that the system can consider the most probable eventualities first and, in the worst-case, ignore less-likely situations if required by resource constraints (see Chapter V). What makes this problem challenging in the context of a complex, dynamic environment is that the probabilities of encountering particular states of the world are dependent not only on the choices of what actions the agent performs, but also by its choices of how quickly it will perform them. The sooner an agent in a dynamic environment detects and responds to a situation, the less opportunity there is for environmental events to intervene.

In this chapter, we provide a foundation for computing state probabilities in a dynamic environment, where the probability of moving from one state to a successor is a function of how long the state persists. In fact, as we point out in what follows, some events could occur in any of a sequence of states, and the time-dependent probability of such an event occurring must take into account the time spent across multiple states, including state-space cycles.

CIRCA-II's probabilistic planning algorithm directly addresses these challenges and is designed to benefit from the compact representation like that found in nondeterministic finite-state automata [42] while also maintaining a probabilistic state-space model like that from a Markov Decision Process (MDP) model [11],[45]. Because there is no "free lunch", the CIRCA-II model is less representationally-efficient than a finite-state machine and may produce suboptimal (but sufficient) plans whereas the MDP would produce plans that maximized overall utility.

This chapter is structured in accordance with the evolution of the CIRCA-II temporal model, including details of how it is used to compute state probabilities and the hard real-time deadlines used to guarantee system safety during plan execution. Because CIRCA-II's planner is based upon the original CIRCA planner, we begin by describing the original non-deterministic state-space model which assumes worst-case properties for all state and action transitions. We also discuss how the recent addition of "reliable" temporal transitions (developed at Honeywell Technology Center) augments this nondeterministic model.

CIRCA-II uses a probabilistic world model to allow statistical knowledge specification and to allow graceful performance degradation (as opposed to the planner failing to return any plan in the original CIRCA) when the Planning Subsystem determines it is impossible to schedule a plan that absolutely guarantees safety during a worst-case execution scenario. Before launching into the details of our model, we present examples to illustrate the types of state-space structures that might be revealed during planning. These examples are intended to clarify terminology used in this chapter and are referenced during subsequent model development. Next, we describe in detail the CIRCA-II probabilistic planning world model, first describing how the user specifies probabilistic state transitions, then working through the equations we have implemented for estimating state probability values from temporal transition probability "rate" functions. We discuss how state probability and timing information is used to guarantee temporal transition to failure *(ttf)* preemption, present an example illustrating the use of our equations, then conclude by with a qualitative comparison of our model to a Markov-based model as well as the original CIRCA nondeterministic approach.

CIRCA's Non-Deterministic Temporal Model

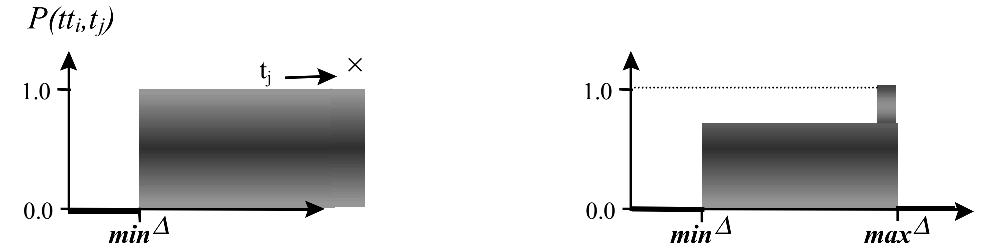
In the original CIRCA [51], the state transition model includes *temporal*, *event*, and *action* transitions. Actions are selected by the planner. If guaranteed, they have a known worst-case execution time so they can preempt all temporal transitions to failure for each state expanded by the planner. Otherwise, their execution time is not computed. Figure 4-1a illustrates the transition timing models used in CIRCA, with time on the x-axis and likelihood of occurring as a function of time since the transition was first active $(P(tt_i,t_j))$ on the y-axis. The value $min\Delta$ represents the time delay before the transition has non-zero probability, while the gray regions represents the "unknown" area in which the transition may or may not occur, but must be considered by the planner. Using this model, event transitions have a $min\Delta$ time of 0, indicating they could occur any time. Temporal transitions have a $min\Delta$ value greater than 0, meaning that, in principle, a hard real-time action might be scheduled (with $deadline \leq min\Delta$) to guarantee transition preemption.

Using this notion of temporal and event transitions, CIRCA is able to build its nondeterministic state model, including all transitions that could occur within the "gray" area of Figure 4-1a. For each state with a matching temporal transition to failure (*ttf*), a "guaranteed" action was chosen, and its deadline was set to the $min\Delta$ value for that temporal transition, thereby preempting the transition and guaranteeing system safety should this state be reached during plan execution. For other states without a *ttf*, any action was considered strictly best-effort to minimize scheduling requirements, so all matching temporal and event transitions were considered possible.

One drawback of the simple nondeterministic transition model from Figure 4-1a is that there is no guarantee a temporal (or event) transition will ever occur. In some cases, a *ttf* may actually be preempted by a *tt* matching the same state (e.g., a climbing aircraft will transition to a new altitude so will not hit distant traffic at its current altitude). The original CIRCA model had no mechanism for representing this possibility, thus was restricted to always finding some action which will preempt the *ttf*, which in some cases is more difficult than letting another *tt* avert the disaster automatically.

Parallel CIRCA research has maintained this basic nondeterministic transition model but added a model for a "reliable" temporal transition as illustrated in Figure 4-1b. This new transition model includes a new time value, *maxΔ*, representing the maximum amount of time that may elapse before the transition <u>must</u> occur, as well as the previous *minΔ* value. Although not all temporal transitions must be assigned a *maxΔ* value, those that are guaranteed to happen before *maxΔ* has elapsed will be able to preempt a *ttf*, thus can be relied upon in states where disaster will be averted without intervention.

Even with the *minΔ* / *maxΔ* model, CIRCA's non-preempted (reachable) state-space is still completely nondeterministic, a structure that does not provide any straightforward measure(s) to use for state prioritization. A major focus of this research is to incorporate approximate knowledge and to facilitate scheduling by removing unlikely states from consideration when required. To achieve much better than a random selection criterion, we must assess the relative importance of states, a quantity difficult to identify in the nondeterministic CIRCA model. In CIRCA-II, we adopt *state probability* as the measure of state importance, so that only improbable states are removed from consideration in "nominal" goal-achievement plans, effectively maximizing the chances that the nominal plan will succeed alone.[20] Additionally, CIRCA-II's planner develops and caches contingency plans to explicitly react to these improbable states should they occur. Below, we describe the CIRCA-II probabilistic temporal model.



a) CIRCA Temporal Transition Model. b) "Reliable" Temporal Transition.

Figure 4-1: Temporal Transition Model for a Nondeterministic State-Space.

State-space Examples

We begin our description of the CIRCA-II probabilistic model with examples to illustrate the various types of state-space structures that may be encountered during planning. The examples are organized from simplest to most complex, and will be referenced again in subsequent sections of this chapter. Note that in all examples, state identifiers (specified as s_k above each state) denote the order in which the states are added to the stack, which is not necessarily the [best-first] order in which they are expanded. Additionally, in all figures, a double arrow denotes a temporal transition while a single bold arrow represents an action transition.

Figure 4-2 shows the most basic state-space structure possible in CIRCA-II. The state space forms a tree, and only two actions are selected: one best-effort for travelling

[20] Probability is not the only possible measure of state importance for CIRCA-II. In Chapter VIII, we discuss future work to incorporate additional parameters (e.g., time horizon, guaranteed task resource usage) along with probabilities for an overall assessment of state importance that will be used when pruning the planner search space.

to the goal location (*set-fix2*) and one guaranteed (*set-land*) to avoid the *crash ttf* following an engine-failure event. This simple example could be accurately handled by the original CIRCA and CIRCA-II. However, the CIRCA-based approach to explicitly scheduling failure-avoidance actions may still be required for this example if plan-execution resources are so limited that both *set-fix2* and *set-land* cannot simultaneously be scheduled into a control plan given action deadline and worst-case execution properties.[21]

Figure 4-3 shows a simple example with a *state-space cycle* between states s_0 and s_2. In this example, again a best-effort action *(set-fix2)* is included to direct the aircraft toward a new location. However, additionally, the system must react *(climb)* if the aircraft descends from a high to low altitude (e.g., during a wind shear event) before it can impact the ground (crash).[22] Intuitively, since the *crash ttf* must be preempted, an aircraft in state s_2 will always transition back to s_0. Thus, due to the persistent state-space cycle with no (non-preempted) exit paths, the system ultimately will transition from s_0 to s_1 with 100% probability, regardless of the exact number of s_0 - s_2 transitions.

Figure 4-4 illustrates the concept of a *dependent temporal transition (dtt).* The basic goal path shown in Figure 4-4 is for the aircraft to fly from s_0 to s_1 to s_3. However, at each state along this path, the same temporal transition (*engine-failure*) may occur. For many world events, the probability of that transition occurring is a function of the amount of time the transition has been continuously active (i.e., matched a state or state sequence). As will be discussed later, the probability of *engine-failure* depends on how

[21] If this is the case, *set-fix2* will only execute when sufficient slack time exists in the CIRCA-II TAP schedule, as described in Chapter I.

[22] As a disclaimer, these examples are not intended to be overly realistic from the control perspective. See Chapter VII for examples of more "realistic" autonomous flight.

long the aircraft engine has been running,[23] thus the CIRCA-II planner must be able to account for apparent "shifting" of the time-dependent *tt* probability functions when they are part of a *dtt* sequence as is illustrated in Figure 4-4. Figure 4-5 depicts a typical state-space that includes multiple cycles and a *dtt* sequence (chain). This example contains no one distinctive feature but is included to illustrate the fact that CIRCA-II's planner must be careful to simultaneously consider the effects of *dtts* and cycles.

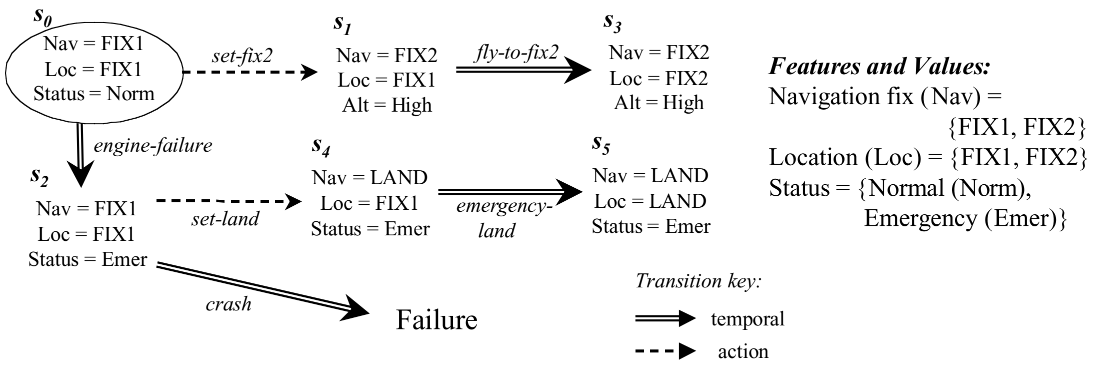


Figure 4-2: Aircraft State-space with Tree Structure.

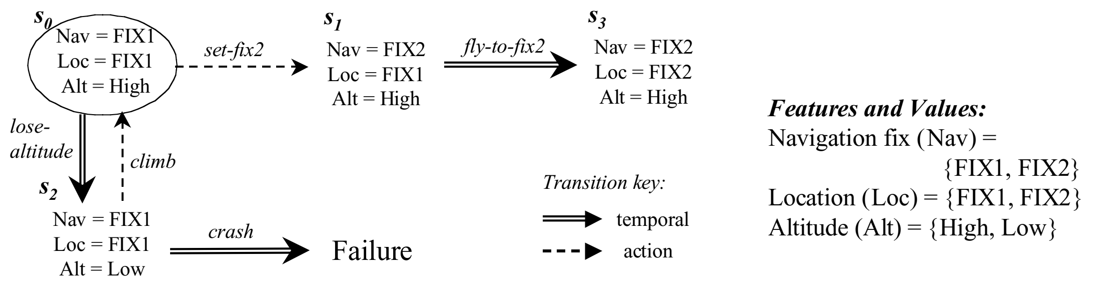


Figure 4-3: Aircraft State-space with a Cycle (s_0 - s_2).

[23] We presume the engine will run continuously during all flights, otherwise we would require a global feature that "remembered" how long the engine had been running. Introducing a global notion of time in CIRCA-II's planner would be analogous to converting an MDP into a non-stationary process, a costly endeavor in either system that we have not yet tackled for CIRCA-II.

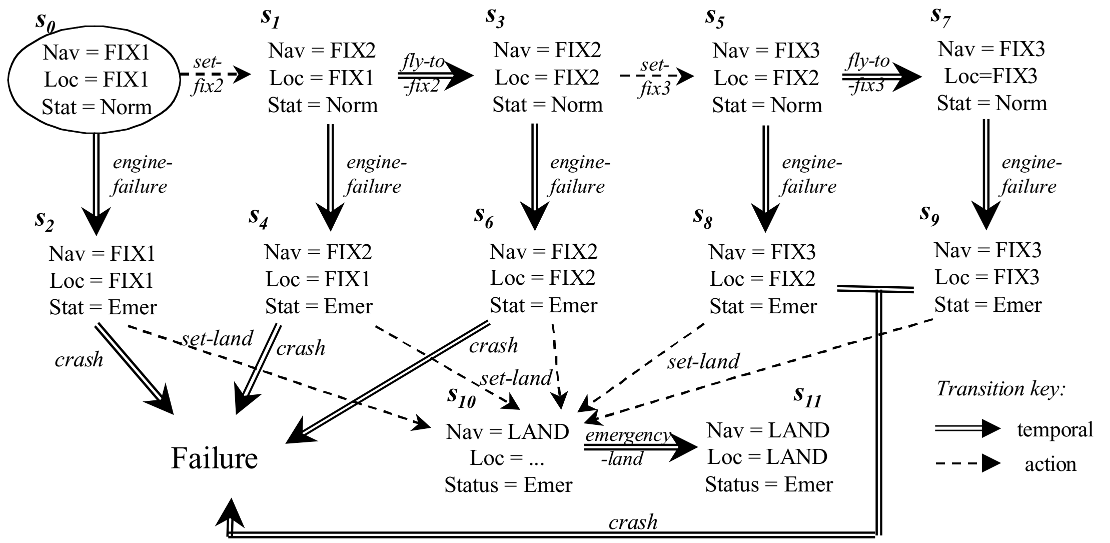


Figure 4-4: Aircraft State-space with a Dependent Temporal Transition *(dtt)*.

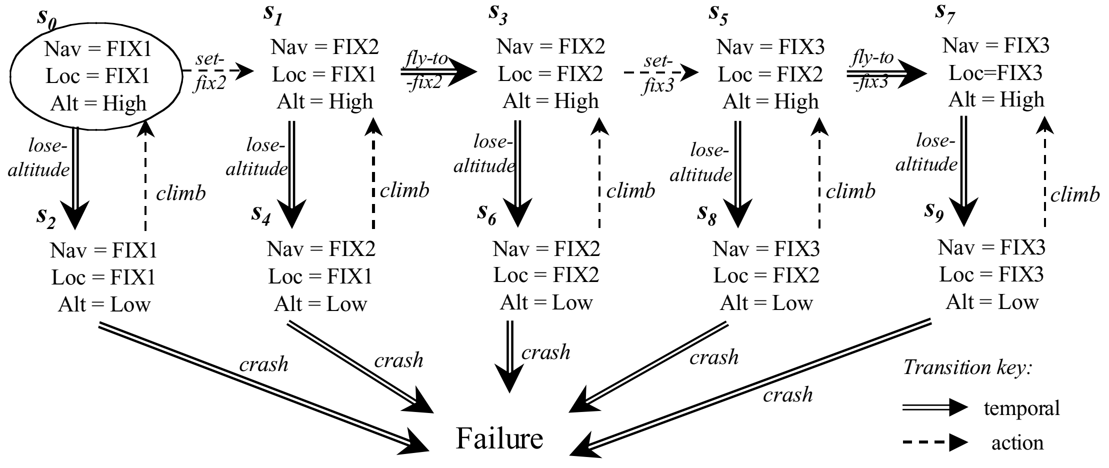


Figure 4-5: Aircraft State-space with Multiple Cycles and a *dtt*.

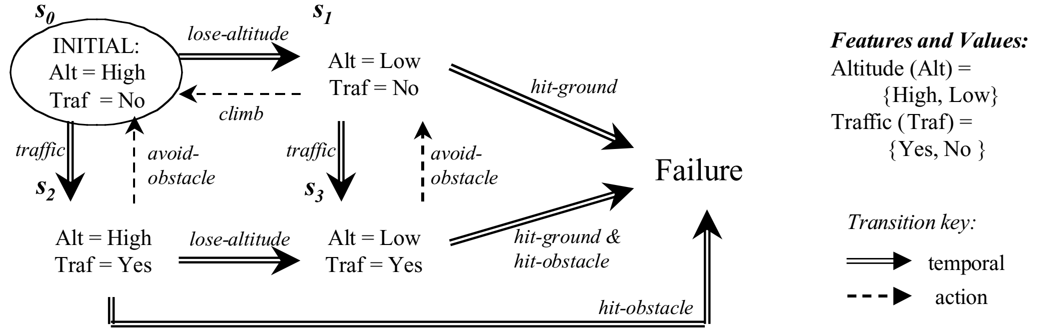


Figure 4-6: CIRCA Planning Anomaly: Multiple Cycles and Dependent *ttfs*.

As a final example, we show a state-space that has to-date not been handled within any implemented version of CIRCA or CIRCA-II.[24] For CIRCA-II, the probability and action-timing-constraint computation algorithms we have incorporated effectively *combine* the timing and probabilistic information from different parents (e.g., s_1 and s_2) so that we can consider *overall* transition cumulative probability values for each state s_k when assessing *ttf* preemption. In the Figure 4-6 case, two *dependent ttfs* (*hit-ground* and *hit-obstacle*) may be simultaneously present in state s_3. Figure 4-6 depicts a valid plan, because all temporal transitions may have sufficient delays before they can occur for preemption with guaranteed actions. However, this plan may not be found by CIRCA-II for certain sets of transition timings due to the combination of dependent transitions and the cycle leading from s_3 back to s_1. Further details regarding this specific state-space structure and our ongoing efforts toward developing a solution for the CIRCA-II stochastic planner are described in Appendix E.

Probabilistic State Transitions

CIRCA-II is built upon the basic algorithms used for CIRCA, so for planning, we keep the same model of a "temporal" and "action" transition set.[25] However, we now account for the fact that individual temporal transitions may occur over time with different and predictable probability distributions. In this section, we describe the specification and utilization of CIRCA-II transition probabilities.

---

[24] Note that the Figure 4-6 example *can* be handled with a non-stationary MDP representation because it does not allow state cycles. Parallel CIRCA research efforts [26] have begun to address this problem by incorporating a model-checking algorithm into the state-space planner.

[25] The term "event transition" was dropped because an event transition can be modeled as a temporal transition with $min\Delta = 0$.

The CIRCA world model is constructed from initial state(s) and a set of state transitions. Action transitions are explicitly controlled by the plan executor thus only occur when selected during planning. All events that cannot be directly controlled are modeled as temporal transitions. When expanding a state, CIRCA finds all temporal transitions with matching preconditions, and selects an action (if any) based primarily on failure avoidance when a *ttf* is present and secondarily on proximity to the goal. The three possible outgoing state transition cases are illustrated in Figure 4-7. Figure 4-7a illustrates the situation where the transition set (action and/or other *tts*) must preempt a *ttf*. We define any transition *tt* out of a state P_{init} as *preempted* when, for each visit, the probability of departing from state P_{init} via *tt* is less than P_{thresh}. Figures 4-7b and 4-7c contain no *ttfs*, so the Figure 4-7b goal-achievement action is strictly best-effort, while the planner has determined that no action enhances goal achievement possibility for P_{init} in Figure 4-7c.

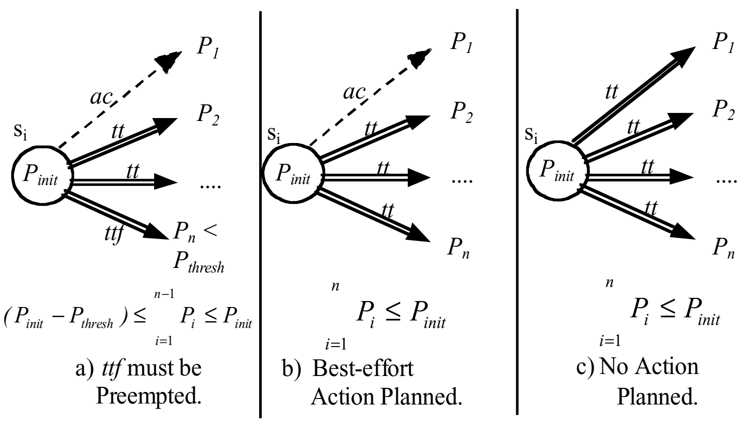

s_i
P_{init}
ac
tt
ttf
P_1
P_2
....
$P_n < P_{thresh}$
$(P_{init} - P_{thresh}) \leq \sum_{i=1}^{n-1} P_i \leq P_{init}$
a) ttf must be Preempted.
$\sum_{i=1}^{n} P_i \leq P_{init}$
b) Best-effort Action Planned.
$\sum_{i=1}^{n} P_i \leq P_{init}$
c) No Action Planned.

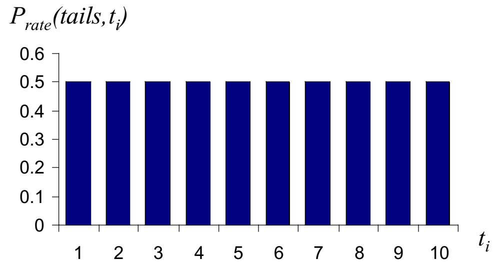
Figure 4-7: Matching Transition Sets for a CIRCA-II State.

As discussed above, the original CIRCA state-space was nondeterministic, designed specifically to allow the planner to guarantee in the worst-case that all *ttfs* would be preempted. In a subsequent CIRCA model [8], cumulative probability functions were attached to each temporal transition, and an algorithm was presented for computing approximate state probabilities. These state probabilities allowed the removal of improbable states when necessary for scheduling purposes, thus enabling the graceful degradation we require for plan scheduling when resources would otherwise be over-utilized. However, this model had several limitations. First, "groups" of temporal transitions all had to match the same set of states; otherwise, the conditional effects of one temporal transition on the cumulative probability of another could not be accurately modeled. This created an unnecessarily large knowledge base. Also, the model from [8] used a strictly local approximation of state probability which did not account for the effects of state-space cycles, new sources to a state already expanded, or dependent temporal transitions *(dtts)*.

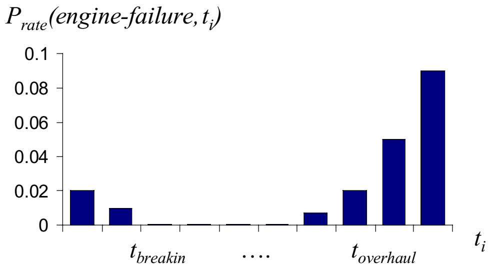
We have captured the "spirit" of the basic CIRCA action/temporal transition model, but the CIRCA-II probabilistic planning process is significantly different from both the original nondeterministic and probabilistic algorithms. The user (i.e., person specifying the planner knowledge base) now describes *discrete-time* temporal transition probabilities as *probability rate* functions $(P_{rate}(tt_j,t_i))$ over a specific time step interval (e.g., seconds). The function $P_{rate}(tt_j,t_i)$ may be described as the likelihood of transition tt_j (i.e., statistical rate at which tt_j occurs) during time step t_i given that tt_j has not already occurred in that state during any earlier time step t_k $(0 \leq k < i)$. This definition is local in the sense that time t_0 corresponds to the time at which the tt_j preconditions were first activated at the current location in the CIRCA-II planner's state-space.

Although the P_{rate} functions presented in this thesis are quite simple, P_{rate} specification is purposely made flexible so that the user can incorporate statistical data and time intervals of any shape or size into a CIRCA-II knowledge base. For example, $P_{rate}(tt_j,t_i)$ may be constructed from an experimentally-derived histogram that describes the likelihood of tt_j in "bins" corresponding to observed time steps [or intervals] t_i at which tt_j occurred. Or, the user may use a *Poisson distribution* [16] for temporal transitions that will occur at some expected time (the mean) but with some uncertainty (the variance). In this dissertation, we have not had sufficient resources to gather accurate statistical data for our knowledge bases. However, we hope to incorporate statistical data in future CIRCA-II knowledge bases for modeling events (e.g., meteorological phenomena) to aid with the "go/no-go" decisions made during flight.

As a simple and intuitive example for $P_{rate}(tt_j,t_i)$, consider a fair coin flipped once per second. The "probability rate" function for a transition from heads to tails has a constant value of 0.5 over each second, because 50% of all coins flipped will land in a "tails" position after exactly one flip, regardless of exactly when that flip occurred with respect to t_0, the time at which this transition was activated. Figure 4-8a shows the probability rate function for this coin flip example.



a) Coin-flip.

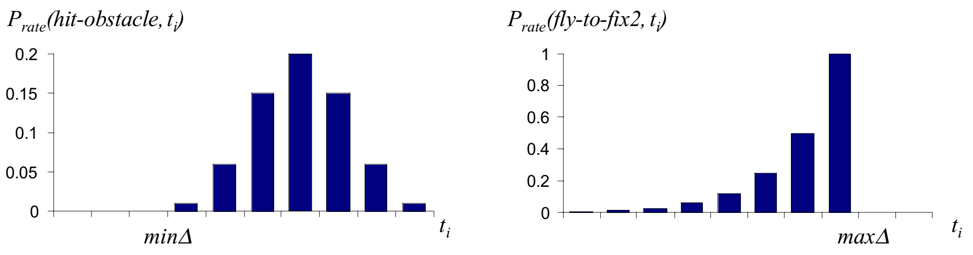


b) Engine-failure.

Figure 4-8: Example Temporal Transition Probability Rate Functions.

As another example, consider the temporal transition "engine failure" for an aircraft. When an engine is first put into service, the "failure rate" decreases during a break-in period, then becomes very small during the normal operation period. When the engine nears the end of its life (e.g., 2000 hours for a small propeller-driven engine), the failure rate increases until the engine is considered "unsafe" and must be retired. An example of the probability rate function for engine failure is shown in Figure 4-8b. We capture such a multi-valued function using a *probability rate table* attached to each CIRCA temporal transition, with probability rate information provided for all time intervals with non-zero probability rate values.

For failure avoidance, the CIRCA-II planner must show that all *ttfs* have a "sufficiently small" probability of occurring before another transition is guaranteed to take the system away from each dangerous state. We define P_{thresh} as the probability threshold value below which a state s_k is effectively considered preempted for each s_k encounter. Figure 4-9a shows a generic trend for a *tt* probability rate function that can possibly be preempted. Consider as an example a transition *hit-obstacle* which matches a state when a collision-course object appears on aircraft radar. Such a *preemptible tt* has an initial period where its probability of occurrence is quite low or even zero (e.g., when the obstacle is still far away), then after some delay the *tt* can occur, with a maximum for this particular example during the time interval in which the obstacle will be closest to the aircraft. In the figure, this delay is labeled *minΔ* in reference to the original CIRCA nomenclature. However, as will be described below, the *conditional effects* of other temporal and action transitions matching each state must be incorporated before computing preemptive transition timing requirements (e.g., the action execution deadline for maneuvering to a "safe" area).



a) “Preemptible” *tt*. b) “Reliable” *tt*.

Figure 4-9: Temporal Transition Probability Rate Function Trends.

Figure 4-9b shows a sample probability rate function corresponding to a *reliable temporal transition*, such as the *fly-to-fix* transitions in the state-space examples presented previously. For such a transition, the probability rate function must eventually increase to 1.0 for some time step, effectively guaranteeing that, if that transition has not occurred earlier, it will occur at this time. Note that both Figures 4-9a and 4-9b are only examples of probability rate functions that could be modeled for preemptible and reliable temporal transitions; other functional forms that satisfy the basic *minΔ/maxΔ* requirements are also possible. For example, if one knows precisely the time step at which a reliable *tt* will occur, the probability rate function can be modeled as 1.0 at that time step and 0.0 everywhere else. As mentioned above, we have used simplistic probability rate functions for the examples presented in this thesis. However, there are no restrictions for their specification so long as consistent time step sizes are used within each knowledge base.[26]

We also require probability rate functions for action transitions so CIRCA-II can assess the relative likelihood of temporal transitions with respect to actions. For guaranteed actions, we know the corresponding state-space transition must occur before or at the action deadline. However, we cannot predict this value more precisely because we do not know where in the cyclic TAP schedule the state requiring the action is first reached. Figure 4-10 illustrates this uncertainty. In the ideal case (labeled *best-case* in the Figure), any state requiring the guaranteed action from TAP4 will be reached just before executing TAP4, in which case the action will complete well before its deadline. In the worst-case, a state requiring the TAP4 action will be reached immediately after the TAP4 test determines the TAP4 action need not execute (labeled *worst-case* in the Figure), in which case the entire schedule must cycle before TAP4 will again execute.

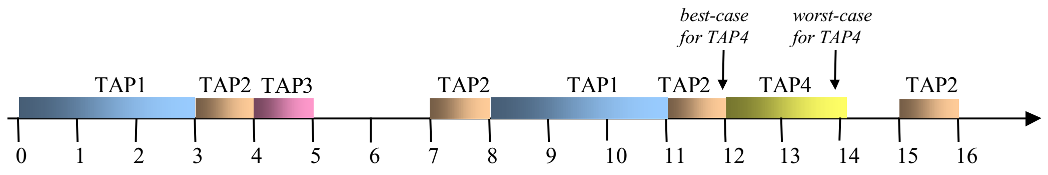


Figure 4-10: Cyclic TAP Schedule in CIRCA-II.

[26] Several test cases have been run on functions ranging from a simple Poisson distribution to periodic, although a periodic probability rate would be rare unless some state features (e.g., plane position around a holding pattern) are not modeled.

Equation 4-1 shows the action probability rate function we adopt to reflect the uncertainty illustrated in Figure 4-10, where *maxΔ(ac)* is the time at which *ac* must have occurred to preempt all *ttfs*. As shown, the probability of the action occurring during each time step (given that it has not yet occurred) increases until it reaches 1 just before *maxΔ*. Graphically, this function is identical to the example "reliable" temporal transition probability rate shown in Figure 4-9b.

$$Pr(ac,t_i) = \begin{array}{lll} \frac{1}{(max\,\Delta(ac)-t_i)} & (t_i < max\,\Delta(ac)) & ? \\ 0 & (t_i \text{ ? } max\,\Delta(ac)) & ? \end{array} \qquad (4\text{-}1)$$

For “if-time” (best-effort) actions, we currently use a constant probability rate function with [relatively small] magnitude assigned by the user in the knowledge base. This is not necessarily an accurate representation. However, the actual best-effort action rate function depends on the TAP schedule (i.e., where the *if-time-server* is inserted, if at all) as well as slack time available during plan execution. In future work, we plan to assign more accurate best-effort action probability rates by iterating between scheduler and planner. We hope to incorporate a probability rate function similar to that for guaranteed actions when the *if-time-server* can be placed into the TAP schedule. Otherwise, we will build the best-effort TAP probability rate function based on average slack time that will be available during plan execution, computed from differences between average and worst-case execution times for the guaranteed TAPs.

State Probability Computation

As described above, all transitions will be assigned probability rate functions that have consistent interval sizes in the CIRCA-II knowledge base. We begin by using these values to compute the relative probability of each temporal and action transition ever occurring from a state, and we presume that all transitions are mutually exclusive because

CIRCA-II contains no model of simultaneously transitioning to multiple states. Then, we use transition probabilities to compute state probabilities. CIRCA-II does not presume any particular structure to the state-space, so we must handle dependent temporal transitions as well as cycles in the state-space. In this section, we describe the equations that form the basis for the probabilistic CIRCA-II temporal model.[27]

Perhaps the most challenging problem with specifying the CIRCA-II stochastic model is accounting for all timing information so that we can have a state-space representation in which individual states are *independent* of time but have incoming and outgoing transitions with probabilities that are *dependent* on time. Table 4-1 defines the set of symbols we will be using to describe state probabilities in CIRCA-II. The planner begins with a set of initial states (each initialized to probability $P_{initial}(s_k)$) and knowledge of all unconditional temporal transition probability rate functions $P_{rate}(tt_j,t_h)$, from which it builds the conditional transition probabilities $P_{cond}(trans_j,t_h,s_k)$ based on other transitions that match state s_k. Then, the CIRCA-II planner uses the resultant cumulative probabilities $P_{cum}(trans_j,s_k)$ to compute child state probabilities $P(s_c)$ from each parent.

The remainder of this section describes the procedure by which transition and child state probabilities are computed/updated during the state expansion process. On startup, the planner is given a set of initial states and assigns them equal probability as defined in Equation 4-2, in which we also set $P_{initial}(s_k)$ to zero for all non-initial states.

$$P_{initial}(s_k) = \begin{array}{ll} \dfrac{1}{n_{initial\_states}} & s_k \lfloor\ initial\_states \\ 0 & s_k \lceil\ initial\_states \end{array} \qquad (4\text{-}2)$$

---

[27] After we discuss the rather involved set of equations used in our state probability and action deadline computations, we present a simple example that works through this entire sequence of computations in an illustrative fashion. We encourage the reader to flip between the equation descriptions and this example frequently to minimize any confusion that may result from this extensive equation set.

Table 4-1: Symbol Definitions for the CIRCA-II Probabilistic Planning Model.

Symbol	Description
tt_i	temporal transition *i*
ac_i	action transition *i*
$trans_i$	transition *i*, including both temporal and any selected action transitions matching the current state
s_k	state *k*
t_h	discrete time step *h*
$n_{initial\_states}$	number of initial states for this plan
$P_{removed}$	Threshold below which states are ignored (removed) due to scheduling constraints.
P_{thresh}	User-defined probability preemption and convergence threshold (default = 0.)
$P_{initial}(s_k)$	Initial state probability contribution to state s_k currently computed assuming a non-informative prior distribution.
$P_{rate}(tt_j,t_h)$	User-defined probability rate function; specifies the probability of transition tt_j occurring during time interval $[t_h,t_{h+1})$ given that transition tt_j has been continuously active for *h* time steps.
$P_{rate}(tt_j,t_h,s_k)$	Unconditional probability of temporal transition tt_j occurring in state s_k during time interval $[t_h,t_{h+1})$.
$P_{rate}(ac,t_h,s_k)$	Unconditional probability of the selected action *ac* occurring in state s_k during time interval $[t_h,t_{h+1})$. This value is 0 if no action has been selected for state s_k.
$P_{rate}(trans_j,t_h,s_k)$	Unconditional probability of transition $trans_j$ occurring in state s_k during time interval $[t_h,t_{h+1})$.
$P_{rate}(none,t_h,s_k)$	Probability of no transition occurring from state s_k during time interval $[t_h,t_{h+1})$.
$P_{dtt}(tt_i,t_h,s_p,trans_j)$	"Shifted" probability rate function to reflect effects of dependent temporal transitions (*dtt*); specifies the probability of temporal transition tt_i occurring during time interval $[t_h,t_{h+1})$ given the current state was reached via transition $trans_j$ from parent state s_p.
$P_{cond}(trans_j,t_h,s_k)$	Conditional probability of transition $trans_j$ occurring in state s_k during time interval $[t_h,t_{h+1})$.
$P_{cum}(trans_j,s_k)$	Cumulative probability of $trans_j$ occurring in state s_k.
$P(s_k,t_h)$	Probability of being in state s_k at time t_h given $P(s_k,t_0)=1$
$P(s_k)$	Probability of reaching state s_k at least once along any state-space path.
$min\Delta(tt_i,s_k)$	Time step at which the cumulative probability of tt_i out of s_k crosses P_{thresh}
$max\Delta\ (ac_{guar},s_k)$	Maximum time step at which ac_{guar} can preempt the fastest *ttf* out of s_k
$deadline(ac_{guar},s_k)$	Deadline (in time steps) for completing ac_{guar} execution after state s_k first becomes active

CIRCA-II incorporates *unconditional* probability rate information for each transition from the knowledge base.[28] Ideally, this rate function could be directly used to approximate conditional probabilities. However, in cases where one or more parent states *(s_p)* match a temporal transition *(tt_i)* that is also active in the current state being expanded *(s_k),* some delay has passed that effectively shifts tt_i 's probability rate function by the amount of time tt_i was active before ever reaching the current state *(s_k)*. When this situation is encountered, we say that tt_i is a dependent temporal transition *(dtt)*, and we must account for this situation when computing conditional probabilities. Equation 4-3 describes the time-dependent probability rate function for transition tt_i in state s_k. This formulation is based on a weighted average of dependent temporal transition shifting effects for tt_i over all parents s_p, including any contribution when s_k is an initial state.

$$P_{rate}(tt_i, t_h, s_k) = \frac{P_{initial}(s_k) P_{rate}(tt_i, t_h) + \underset{\forall(p,j)\ni \overline{s_p}\ ?\ \overset{trans_j}{?\,?}\ \blacklozenge\ s_k \surd}{} P(s_p) P_{cum}(trans_j, s_k) P_{dtt}(tt_i, t_h, s_p, trans_j)}{P_{initial}(s_k) + \underset{\forall(p,j)\ni \overline{s_p}\ ?\ \overset{trans_j}{?\,?}\ \blacklozenge\ s_k \surd}{} P(s_p) P_{cum}(trans_j, s_k)} \quad (4\text{-}3)$$

Equation 4-4 describes how a single parent state *(s_p)* affects the probability rate function in state s_k. If parent s_p does <u>not</u> match tt_i, then the original probability rate function is passed as P_{dtt} to Equation 4-3. Otherwise, if parent s_p <u>does</u> match tt_i, then we shift the probability rate function by the amount of time it has taken parent s_p to transition via transition $trans_j$ to s_k. Since we only have a probabilistic representation of the amount of time required to transition from s_p to s_k (P_{cond} term used in Equation 4-4), we again use a weighted average formulation to express the shift of the probability rate function, normalizing by the cumulative probability of the s_p s_k transition. Equation 4-4 takes

[28] Although we refer to all P_{rate} functions as unconditional, a more precise way to define these functions is conditional on nothing else happening to the state, and we seek to incorporate the effects of other transitions into our conditional probability P_{cond} estimate.

into account the case where a sequence of states all match the same *dtt* since the P_{rate} function for s_p has already incorporated this information. When a *dtt* is present in all states around a cycle that includes states s_p and s_k, the relative effect of this "shift" should be included in the overall rate function for s_k. CIRCA-II performs a depth-first search from transitions out of s_k and its descendants to identify and account for their overall *dtt* contribution effectively as separate "parents" s_p' of s_k.[29]

$$P_{dtt}(tt_i, t_h, s_p, trans_j) = \begin{cases} P_{rate}(tt_i, t_h) & \text{when} \quad tt_i \notin \{trans(s_p)\} \\ \sum_{g=0} \dfrac{P_{cond}(trans_j, t_g, s_p) P_{rate}(tt_i, t_g + t_h, s_p)}{P_{cum}(trans_j, s_p)} & \text{when} \quad tt_i \in \{trans(s_p)\} \end{cases} \tag{4-4}$$

We have just defined a state-dependent *tt* probability rate function that accounts for all *dtts* in a state s_k. In our model, we assume that we will never require dependent action transitions, since we control their execution properties. Thus, for actions, the probability rate function is either the "reliable probability rate" expression described in Equation 4-1 for guaranteed actions or else the user-defined constant for best-effort actions.

A basic premise of our probability model is that we wish to minimize knowledge base size by allowing the user to specify conditionally-independent state transitions.[30] In our model, multiple transitions may match a state and we assume that these events will be

[29] Currently, this search terminates after the first cycle is expanded (i.e., each cycle is only traversed once for *dtt* computations). In the future, this algorithm will be modified to terminate only when the probability addition due to repeated cycle traversals converges.

[30] In cases where the postcondition features and/or the *tt* probability rate tables change markedly due to other world event(s), the knowledge base creator must be careful to manually insert conditional dependencies that will not be computed by our model. We have not encountered such a situation to-date but acknowledge that it could exist.

mutually exclusive.[31] The CIRCA-II planner approximates conditional probability rate functions *(P_{cond})* from independent probabilities *(P_{rate})* using a model similar to the "noisy-OR" formulation defined for Bayesian Networks [61]. First, we compute the probability that no transition occurs during time step t_h in state s_k, as shown in Equation 4-5. Next, we approximate the conditional probability of transition $trans_j$ occurring during time step t_h from state s_k. As shown in Equation 4-6, P_{cond} is based on P_{rate} (from Equation 4-3) and is weighted by the probability that some transition occurs during time step t_h normalized by the sum of the P_{rate} functions for all transitions matching s_k.

$$P_{rate}(none, t_h, s_k) = \prod_{\forall trans_r \in trans(s_k)} \left(1 - P_{rate}(trans_r, t_h, s_k)\right) \tag{4-5}$$

$$P_{cond}(trans_j, t_h, s_k) = \frac{P_{rate}(trans_j, t_h, s_k)\left(1 - P_{rate}(none, t_h, s_k)\right)}{\sum_{\forall trans_r \in trans(s_k)} P_{rate}(trans_r, t_h, s_k)} \tag{4-6}$$

After CIRCA-II computes P_{cond} for all transitions out of s_k, the planner is finally ready to compute state probability information. As a first step, Equation 4-7 shows the probability of remaining in state s_k given that the system has entered s_k just prior to time step t_0. This recursive computation is based exclusively on the probability that no transition takes the system out of state s_k at time step t_h, scaled by the probability that the system has not already left state s_k prior to t_h.

$$P(s_k, t_h) = \begin{cases} 1 & \text{when } h = 0, \\ P(s_k, t_{h-1}) P_{rate}(none, t_{h-1}, s_k) & \text{when } h > 0. \end{cases} \tag{4-7}$$

---

[31] If a combination of transitions could happen simultaneously, CIRCA-II accounts for this by allowing one transition to occur then calculating that the next transition occurs immediately *($min\Delta=0$)*.

Next, Equation 4-8 describes the computation of overall cumulative probability for each transition $trans_j$ from s_k. Since a planner cannot actually sum to infinite time, we define a convergence criterion in Equation 4-9 which will be met whenever either P_{cond} for $trans_j$ decreases to near-zero (as defined by P_{thresh}) or else the likelihood of still being in state s_k has diminished to near-zero.

To maximize computational efficiency, the CIRCA-II planner computes the set of quantities described by Equations 4-3 through 4-8 for each time step t_h before moving to the next time step t_{h+1}. Equation 4-4 (for P_{dtt}) also included a summation to infinity, but the entire computation sequence will automatically terminate when the planner determines that P_{cum} has converged.

$$P_{cum}(trans_j, s_k) = \sum_{h=0}^{\infty} P_{cond}(trans_j, t_h, s_k) P(s_k, t_h) \tag{4-8}$$

$$P_{cum} \text{ converged at } \{t_c \ni \forall(t_h > t_c)(P_{cond}(trans_j, t_h, s_k) < P_{thresh}) \vee (P(s_k, t_h) < P_{thresh})\} \tag{4-9}$$

Our ultimate goal is to compute time-independent state probability values *($P(s_k)$)*. Because of CIRCA-II's cyclic state-space, we have incorporated a matrix algorithm based on theoretical constructs from [36] to compute CIRCA-II state probabilities.[32] In this paragraph, we summarize the algorithm used to convert P_{cum} values from Equation 4-8 to state probabilities $P(s_k)$ for all states s_k identified thus far in the planner's state-space search. Let the matrix $\boldsymbol{M}$ represent the current planner state-space, where each element m_{kl} is the cumulative probability $P_{cum}(trans_j,s_k)$ of transitioning from state s_k via $trans_j$ to child state s_l. $\boldsymbol{M}$ is partitioned as shown in Figure 4-11 such that the first r rows and columns contain only *absorbing nodes* (i.e., states that either have not yet been expanded

[32] The matrix algorithm for handling cycles during probability computations was motivated by the MDP literature and implemented by Haksun Li. An evaluation of this matrix algorithm and overall CIRCA-II probabilistic planner accuracy is provided in [43].

or that are expanded but contain no outgoing transitions), while the last *(n-r)* rows and columns are *transient nodes* (i.e., expanded states with outgoing transitions). As described in [36], the probabilities of transitioning from any node in the transient set to any node in the absorbing set is given by the matrix ***P*** in Equation 4-10, while Equation 4-11 then shows the computation of $P(s_l)$ for each absorbing node.[33]

$$\begin{array}{c|c|c} & \multicolumn{2}{c}{\text{Child state } s_l} \\ \text{Source state } s & 1 \quad \cdots \quad r & \cdots \quad n \\ \hline 1 \;\cdots\; r & \begin{array}{c}\text{I}\\ \text{(Identity)}\end{array} & \begin{array}{c}0\\ \text{(Zero)}\end{array} \\ \hline \cdots \; n & \begin{array}{c}\text{R}\\ \text{(Transient to Absorbing)}\end{array} & \begin{array}{c}\text{Q}\\ \text{(Transient to Transient)}\end{array} \end{array}$$

Figure 4-11: Matrix ***M*** used for CIRCA-II State Probability Computation.

$$\mathbf{P} = (\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R} \tag{4-10}$$

$$\forall s_l \lfloor \; absorbing\_states \quad P(s_l) = \underset{s_k \lfloor \; initial\_states}{p_{kl}} \tag{4-11}$$

The Figure 4-11 construction of ***M*** gives $P(s_k)$ for all absorbing nodes (states). To compute transient state probabilities, we rebuild ***M*** for each transient state s_{trans} with all outgoing edges truncated. Then, we recompute ***P*** from Equation 4-10 and sum all initial state probabilities into s_{trans} to give $P(s_{trans})$, as shown for ***M*** in Equation 4-11. As discussed further in [43], truncation of outgoing edges for transient state probability computation is acceptable because, for all states s_k, we define $P(s_k)$ is the probability of visiting s_k at least once (see Table 4-1), thus multiple visits (i.e., by traversing around a cycle from s_k back to s_k) need not contribute.

[33] For notational simplicity, we use p_{kl} to represent the element of matrix ***P*** corresponding with the probability of transitioning (via one or more transitions) from s_k to s_l. However, ***P*** must be ordered to partition the transient and absorbing nodes, thus state s_l (where *l* corresponds to the order in which CIRCA-II created the state) need not be matrix row *l*.

Failure-Avoidance Guarantees

To assure guaranteed real-time failure avoidance, the CIRCA-II planner must be capable of reasoning about *ttf* preemption with either a guaranteed action or a reliable *tt* (or a sequence of actions and/or reliable *tts*). In the original CIRCA, we presumed that the user had defined precise values for *min*Δ that would allow the planner to set the action deadline (separation constraint for the single-processor scheduler) to the smallest *ttf min*Δ that action must preempt, or else verified that a reliable *tt* (with pre-specified *max*Δ) automatically preempted all *ttfs*. Unlike with the original CIRCA, we cannot *a priori* define a specific time *min*Δ at which a *ttf* may occur because we now utilize a probability model that necessitates *dynamic* computation of state probabilities over time. We also need an analogous representation of *maxΔ* for guaranteed action and reliable temporal transitions to ensure preemption.

In keeping with the original CIRCA terminology as much as possible, we define the time step *minΔ* for a tt_i (or ttf_i) out of state s_k as that time step at which the cumulative probability for tt_i crosses (or reaches) the preemption threshold P_{thresh}. Equation 4-12 shows our *minΔ* (time step t_m) definition, given by the conditions that the cumulative probability up to t_{m-1} must be less than P_{thresh} but at t_m must be greater than or equal to P_{thresh}. In the current CIRCA-II implementation, we only use Equation 4-12 for *ttf* preemption computations, but give a more general definition here because other *tt*'s can theoretically be preempted as well.

$$min\,\Delta(tt_i, s_k) = t_m \ni \left[\sum_{h=0}^{m-1} P_{cond}(tt_i, t_h, s_k) P(s_k, t_h) < P_{thresh}\right] \wedge \left[\sum_{h=0}^{m} P_{cond}(tt_i, t_h, s_k) P(s_k, t_h) \geq P_{thresh}\right] \quad (4\text{-}12)$$

Our ultimate goal is to effectively push *minΔ* for the set of *ttfs* to infinity. In other words, we never want the cumulative probability of any *ttf* out of any reachable state s_k to cross P_{thresh}. All reachable states s_k with one or more matching *ttfs* must also have a "reliable set" of action and/or temporal transitions (excluding the *ttfs*) that are guaranteed to occur prior to any *ttf*.[34] CIRCA-II dynamically computes transition, state, and *ttf* $min\Delta$ values at each time step (starting with t_0). After time step t_f during which the probability of remaining in state s_k ($P(s_k, t_h)$ computed previously in Equation 4-7) drops below P_{thresh}, we consider our computations converged. If no $min\Delta$ has been defined for any *ttf* upon reaching t_f, we know all *ttfs* have successfully been preempted since we have already departed state s_k via some other transition. For cases in which the set of *tts* alone are sufficient to preempt all *ttfs* in s_k, we say the *ttfs* have been preempted by a "reliable set of *tts*". Otherwise, CIRCA-II must select and compute timing information for a guaranteed action to allow preemption of all *ttfs* in s_k. Once the action is selected (details of which are discussed in Appendix A), hard real-time constraints for this action must be computed to guarantee *ttf* preemption. Equation 4-13 describes $max\Delta(ac_{guar},s_k)$, defined as the maximum number of time steps that may elapse between first entering state s_k and <u>safely</u> exiting s_k. This value is set to the minimum $max\Delta$ for all *ttfs* in state s_k and is used by the planning "post-processor" for computing action real-time deadlines.

$$max\,\Delta(ac_{guar},s_k) = \{min\,\Delta(ttf_i,s_k) \ni min\,\Delta(ttf_i,s_k) \le min\,\Delta(ttf_j,s_k)\ \forall ttf_i,ttf_j \lfloor\ ttf(s_k), i\ ?\ j\ \} \quad (4\text{-}13)$$

Once planning is complete, we have to compile all information for the guaranteed actions to pass along to the CIRCA-II scheduler. Equation 4-14 defines the action deadline (equal to *separation constraint* for the single-processor scheduler) that will be passed along to the scheduler. These deadlines are computed separately for all $ac_{guar} \in$ *{planned, guaranteed action set}*. We require that each ac_{guar} be assigned the worst-case (minimum) $max\Delta$ to guarantee preemption of all *ttfs* from all $s_k \in$ *{reachable state set S}*

[34] The nondeterministic CIRCA planner requires <u>one</u> reliable *tt* or a guaranteed action to preempt a *ttf*. We now are able to measure the conglomerate effects of multiple transitions acting on s_k. Thus, we are able to use multiple *tts* (along with an action if required), each of which alone would not be able to assure preempting the s_k *ttfs*, but which <u>together</u> preempt all *ttfs* out of reachable state s_k.

for which ac_{guar} is planned. Then, we must subtract the action worst-case execution *(wcet)* so that ac_{guar} actually completes execution prior to its deadline.[35]

$$deadline(ac_{guar}) = min_{s}(max\Delta(ac_{guar}, s_k)) - wcet(ac_{guar}) \tag{4-14}$$

Figure 4-12 summarizes the probabilistic planning algorithm used for CIRCA-II. Much like in the original CIRCA, state expansion is performed with actions selected for each state as required. The overall loop of the algorithm (Steps 2-8) occurs for each reachable state s_k expanded by the planner. The inner loop (Steps 4-6 in Figure 4-12) performs a binary search on the guaranteed action $max\Delta$ for state s_k to determine the maximum allowable value that can still preempt the s_k *ttfs*. As will be discussed in subsequent chapters, we utilize our state probability model not only to select actions and assign their hard-real time deadlines, but also to assign relative *priority values* to states and the actions planned for them and incorporate algorithms to *remove unlikely states* from consideration when scheduling the preemptive actions required to guarantee absolute failure avoidance is impossible.

Several situations can occur that necessitate backtracking during planning. First, we may discover a state from which all *ttfs* cannot be preempted (e.g., no action is "fast enough"). In this case, we perform dynamic backtracking [24] through the sources to that state in an attempt to either preempt transitions to this state or else choose different actions which do not lead to this state.[36] The basic implementation of dynamic backtracking in CIRCA-II is discussed in [4].

[35] If the Equation 4-14 computation results in a negative deadline, the planner must backtrack and select an alternative action with smaller *wcet*.

[36] The original CIRCA performed chronological backtracking and added such a state to a "blacklist" avoided during future state expansion. Our incorporation of path-based state probability updating has resulted in this migration toward dynamic backtracking.

```
1.  Put initial state set on state stack to be expanded.
2.  Pop next highest-probability state $s_k$ off stack for expansion.
3.  Select action for state $s_k$ (if any).  [See Appendix A for details.]
    If there is no ttf from $s_k$, classify action as "best-effort".
    Otherwise, classify the action as "guaranteed" and assign it a
    preliminary maxΔ value of infinity.
4.  Compute all outgoing transition cumulative probabilities ($P_{cum}$) up to
    convergence time $t_c$.
5.  If at least one ttf minΔ has been defined prior to $t_c$, store the
    current action maxΔ as $maxΔ_{old}$ then reset maxΔ per Equation 4-13.
    Go to Step 4.
6.  If ($maxΔ_{old}$ - maxΔ) > $n_{converged}$ time steps (a user-specified value),
    increase maxΔ to ($maxΔ_{old}$ - maxΔ)/2. Go to Step 4.
7.  Create new offspring states when they do not already exist; add new
    or modified states back on the state stack for further expansion.
8.  Compute state probabilities using the matrix algorithm for both
    absorbing and transient states.
9.  While more states require expansion, go to Step 2.
```

Figure 4-12: CIRCA-II Probabilistic Planning Algorithm.

Another situation that must be addressed via backtracking is that in which a dependent temporal transition chain exists with a *ttf*. Our algorithms will work without backtracking if a sequence of "reliable" temporal transitions (or a sequence of "reliable" temporal transitions terminating with one guaranteed action) preempt a dependent *ttf* in all states in the sequence. However, because our algorithms are set up to automatically maximize action *maxΔ* and consider only local preemption requirements (i.e., guaranteeing that no *immediate* descendants of expanded state s_k are failure states), any immediate descendant state s_d of s_k that results from a guaranteed action and matches the same *ttf* will see an unrealistically small *minΔ* value for this *ttf* unless some other *tt* for s_d fortuitously occurs with near-100% probability within the first few time steps. When such a situation currently occurs, we backtrack to the previous state and decrease the *maxΔ* value used until it is sufficiently small.[37] We are in the process of designing a

[37] We currently do not optimize this value, so *maxΔ* may be set to a smaller value than is necessarily required. This condition is safe but makes the guaranteed action more difficult to schedule than might otherwise be necessary.

global methodology for computing *maxΔ* with dependent *ttfs*, along with more efficient backtracking processes both for guaranteed *ttf* avoidance and goal achievement.

State Probability and Deadline Computation Example

In this section, we provide an example that illustrates the use of the above probability model for computing state probabilities and action deadlines during planning. Figure 4-13 shows the first planning step. Initial state s_0 has two outgoing *tts* (*lose-altitude* and *fly-to-fix3*) but is safe because neither is a *ttf*. Our first goal is to estimate the conditional probabilities of these transitions from their rate functions. As given by Equation 4-3, since neither temporal transition is a *dtt*, the rate functions for the *tts* out of state s_0 are unmodified $(P_{rate}(tt_i,t_h, s_0) = P_{rate}(tt_i,t_h))$ and are illustrated in Figure 4-14 (note the different axis scaling in the Figure). We use Equations 4-5 and 4-6 to compute conditional probabilities $P_{cond}(tt_i,t_h, s_0)$ for both *tts*, with the result plotted in Figure 4-14.

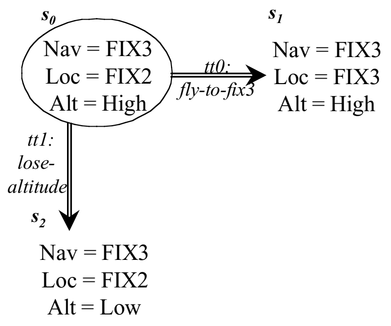


Figure 4-13: Expansion of Initial State s_0.

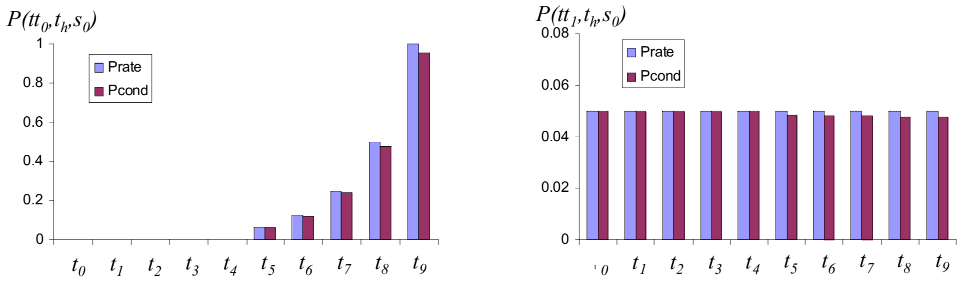


Figure 4-14: s_0 Transition Probabilities.

Next, we compute the cumulative probabilities for the transitions from Equations 4-7 through 4-9. Table 4-2 shows the computation of conditional and cumulative probabilities for the Figure 4-13 example. The final (converged) values for cumulative probability are highlighted in the Table.

Table 4-2: Transition Probabilities out of s_0.

Time step (h)	$P_{rate}(tt_0,t_h)$	$P_{rate}(tt_1,t_h)$	$P_{none}(s_0,t_h)$	$P_{cond}(tt_0,t_h,s_0)$	$P_{cond}(tt_1,t_h,s_0)$	$P(s_0,t_h)$	$P_{cum}(tt_0,s_0)$ to t_h	$P_{cum}(tt_1,s_0)$ to t_h
0	0.000	0.050	0.950	0.000	0.050	1.000	0.000	0.050
1	0.000	0.050	0.950	0.000	0.050	0.950	0.000	0.098
2	0.000	0.050	0.950	0.000	0.050	0.903	0.000	0.143
3	0.000	0.050	0.950	0.000	0.050	0.857	0.000	0.185
4	0.000	0.050	0.950	0.000	0.050	0.815	0.000	0.226
5	0.200	0.050	0.760	0.192	0.048	0.774	0.149	0.263
6	0.250	0.050	0.713	0.240	0.048	0.588	0.289	0.292
7	0.333	0.050	0.633	0.319	0.048	0.419	0.423	0.312
8	0.500	0.050	0.475	0.477	0.048	0.265	0.550	0.324
9	1.000	0.050	0.000	0.952	0.048	0.126	**0.670**	**0.330**

The state probabilities may now be computed for s_1 and s_2. For this system, there are one transient state (initial state s_0 with probability $P(s_0) = 1.0$) and two absorbing states, s_1 and s_2. For this simple example, one may immediately observe that the absorbing state probabilities correspond to the cumulative probabilities *($P(s_1) = 0.67$ and $P(s_2) = 0.33$)*. In CIRCA-II, these same values are computed using the matrix algorithm, which we will utilize in more detail below.

After computing s_1 and s_2 state probabilities, the CIRCA-II planner expands the next most-probable state, s_1. For this example, assume no outgoing transitions match s_1, so s_1 remains an absorbing state even after expansion. The planner then expands state s_2, which matches one *ttf (crash)* as shown in Figure 4-15. The planner selects an action *(climb)* that must be guaranteed to preempt the *ttf*, and iterates to compute a *maxΔ* time of 5 to preempt the *ttf* with 100% certainty. The probability rate functions for the transitions out of s_2 are shown in Figure 4-16. Note that the conditional probabilities are identical to the unconditional rate functions because the two transitions never simultaneously have non-zero probabilities during any time step. As with s_0, we now compute s_2 transition cumulative probabilities, shown with all supporting calculations in Table 4-3.

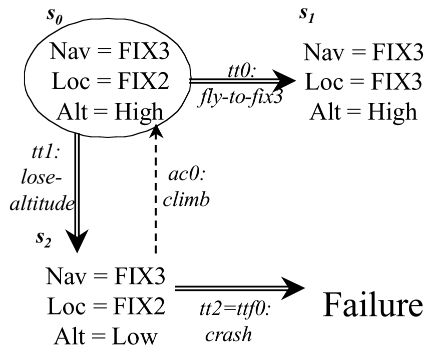
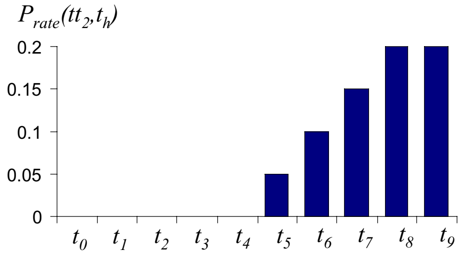


Figure 4-15: Expansion of State s_2.



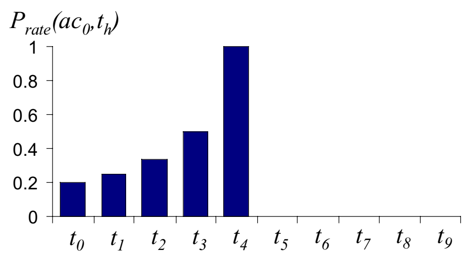


Figure 4-16: s_2 Transition Probabilities.

Table 4-3: Transition Probabilities out of s_2.

Time step (h)	$P_{rate}(tt_2,t_h)$	$P_{rate}(ac_0,t_h)$	$P_{none}(s_0,t_h)$	$P_{cond}(tt_2,t_h,s_0)$	$P_{cond}(ac_0,t_h,s_0)$	$P(s_0,t_h)$	$P_{cum}(tt_2,s_0)$ to t_h	$P_{cum}(ac_0,s_0)$ to t_h
0	0.00	0.20	0.80	0.00	0.20	1.00	0.00	0.20
1	0.00	0.25	0.75	0.00	0.25	0.80	0.00	0.40
2	0.00	0.33	0.67	0.00	0.33	0.60	0.00	0.60
3	0.00	0.50	0.50	0.00	0.50	0.40	0.00	0.80
4	0.00	1.00	0.00	0.00	1.00	0.20	0.00	1.00
5	0.05	0.00	0.95	0.05	0.00	0.00	0.00	1.00
6	0.10	0.00	0.90	0.10	0.00	0.00	0.00	1.00
7	0.15	0.00	0.85	0.15	0.00	0.00	0.00	1.00
8	0.20	0.00	0.80	0.20	0.00	0.00	0.00	1.00
9	0.20	0.00	0.80	0.20	0.00	0.00	**0.00**	**1.00**

CIRCA-II now computes state probabilities using the matrix algorithm. The matrix **M** containing all edges shown in Figure 4-15 is shown in Figure 4-17, partitioned such that the absorbing states appear in rows/columns before transient states. Also shown in Figure 4-17 is the **P** matrix computed via Equation 4-10. The probabilities of absorbing states s_1 and failure *(f)* are then computed from the initial state (s_0) row of **P**, giving final values $P(s_1)=1.0$ and $P(f)=0.0$. When truncating the outgoing edges from s_2

and recomputing **P** with s_2 as an absorbing state, we observe that the probability of state s_2 remains $P(s_2)=0.33$, an intuitive result since we have truncated all edges that were not present during the first state expansion step from Figure 4-13.

$$\begin{array}{c|cc|cc}
 & & \text{Child state} & & \\
\mathbf{M} & s_1 & f & s_0 & s_2 \\
\hline
s_1 & 1 & 0 & 0 & 0 \\
f & 0 & 1 & 0 & 0 \\
\hline
s_0 & 0.67 & 0 & 0 & 0.33 \\
s_2 & 0 & 0 & 1 & 0
\end{array}
\quad \text{Source state}
\qquad\qquad
\begin{array}{c|cc}
\mathbf{P} & s_1 & f \\
\hline
\boldsymbol{s_0} & \mathbf{1} & \mathbf{0} \\
s_2 & 1 & 0
\end{array}
\leftarrow \text{Initial to absorbing state row}$$

Figure 4-17: **M** and **P** Matrices used for State Probability Computations.

The state expansion process illustrated by this example continues until all states have been expanded and all state probabilities have been computed. If the resulting action set is successfully scheduled, plan development is complete. Otherwise, tradeoffs are required for development of a schedulable plan. In this dissertation, we primarily consider a procedure by which we dynamically modify a state probability threshold $P_{removed}$ below which states are ignored, thereby facilitating scheduling by reducing the set of states for which *ttf* preemption must be guaranteed.[38]

For this example, we assumed a value $P_{thresh} = 0.0$, where P_{thresh} is the probability threshold below which individual *ttfs* are considered preempted. Increasing the value of P_{thresh} will, in many cases, also allow an extension of guaranteed action $max\Delta$ (thus deadline) value. Table 4-4 shows the cumulative probabilities for ttf_0 out of state s_2 for different $max\Delta$ values between 5 and 10. If, for example, P_{thresh} were set to 0.05, corresponding to a 5% chance of system failure each time state s_2 is reached, the ac_0

[38] We do not explicitly place limits on $P_{removed}$ in this work, so we are always able to eventually generate a schedulable plan with a large $P_{removed}$ value. In the future, CIRCA-II should automatically decide whether to work toward its goals versus refuse to even enter its environment when a substantial set of states (with high $P_{removed}$) are ignored.

$max\Delta$ could be relaxed from 5 to 7, or with P_{thresh} set to 0.2, the original *$max\Delta$* could be doubled to 10, effectively cutting in half the scheduling resources required for this action. In future work, we will augment the CIRCA-II planner to reason about the tradeoffs associated with the simultaneous relaxation of both $P_{removed}$ and P_{thresh}.

Table 4-4: *Crash (ttf_0)* Probability from s_2 for Various *Climb (ac_0) $max\Delta$* Values.

ac_0 max^{Δ}	$P_{cum}(tt_2 = ttf_0, s_2)$
5	0
6	0.01
7	0.032
8	0.069
9	0.123
10	0.187

Comparative Evaluation of the CIRCA-II Temporal Model

We have not yet quantitatively evaluated the probabilistic temporal model for CIRCA-II presented in this chapter, except for an analysis of the sources and relative magnitudes of inaccuracies resulting from our "weighted average" approach (see [43]). In this section, we discuss overall properties of our model *qualitatively* by comparing aspects of the CIRCA-II model to the original CIRCA and MDP models. Figure 4-18 shows a qualitative plot of system success probability[39] *($P_{success}$)* vs. the inverse of resource capacity *(c)*. In this plot, we presume there is at least one set of actions that can preempt all *ttfs* given sufficiently large plan-execution resource capacity. As might be expected, in the original CIRCA (depicted by the bold line in Figure 4-18), there is a 100% probability of success so long as a valid plan can be scheduled on *c*, but when scheduling fails for any action set that preempts all *ttfs*, no tradeoffs are available and the probability of success immediately drops to 0%.

[39] Due to our bias toward "safety first", we measure "success" in terms of failure-avoidance instead of goal-achievement.

With CIRCA-II, the probability of success depends on the setting of P_{thresh} and also the setting of a "removed-state" probability threshold $(P_{removed})$ which will be discussed in the next chapter. In Figure 4-18, we show the qualitative expected performance of both CIRCA and CIRCA-II as resource capacity decreases (i.e., *1/c* increases). The bold curve represents CIRCA's performance. With $P_{thresh}= P_{removed} =0.0$, CIRCA-II performance emulates that of the original CIRCA, since only absolute preemption is sufficient. The solid curve set in Figure 4-18 represents CIRCA-II performance for varying P_{thresh} with $P_{removed}$ =0.0. As shown, with even a small but non-zero P_{thresh}, the system may show marked adaptation to reduced resource capacity. This is in part because the CIRCA-II planner can utilize multiple temporal and action transitions to cooperatively preempt *ttfs*, such that there is at least a $(1- P_{thresh})$ probability of having exited each state s_k via some transition other than a *ttf*. As P_{thresh} increases, the probability of success decreases but the system is able to accommodate a smaller resource capacity for the same domain complexity.

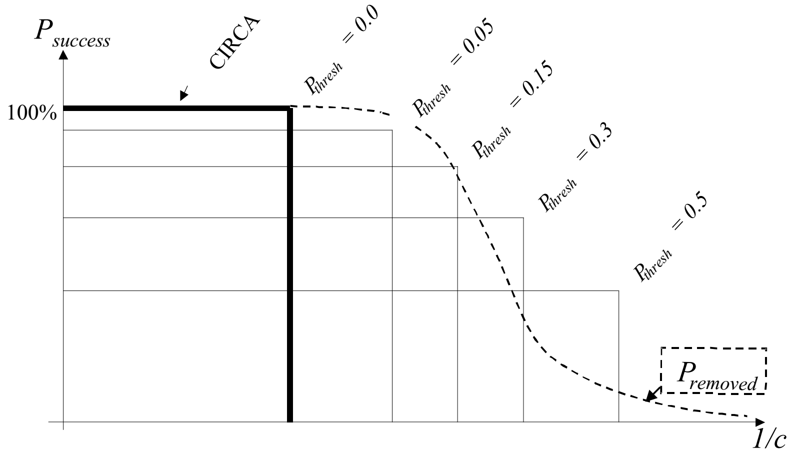


Figure 4-18: Success Probability vs. Inverse Plan-Execution Resource Capacity.

The user currently specifies the constant P_{thresh} before CIRCA-II begins. However, CIRCA-II does automatically adjust a threshold $P_{removed}$ below which states are not considered in the current plan being developed to improve schedulability. The dashed curve in Figure 4-18 shows how $P_{removed}$ adaptation affects the success likelihood within a single plan. Although this curve does not fit a specific set of generated data points, we do expect a sigmoid shape to this curve, since at the limits, the success probability will be 100% when sufficient resources are available (again emulating CIRCA) and 0% when too few resources are available for executing any actions. As shown in the Figure, points along the curve for varying $P_{removed}$ need not precisely align with identical P_{thresh} values since they are utilized differently by the planner (see details for $P_{removed}$ in Chapter V).

In the Figure 4-18 plot, we only compare CIRCA and CIRCA-II "success" rates because an MDP planner would need to be cast in a CIRCA-like framework to actually produce the "guaranteed" task schedule for us to actually assess $P_{success}$. In future work, regardless of MDP complexity, we would like to better study the fit of an MDP planner into CIRCA-II to analyze its ability to generate plans that gracefully degrade as plan-execution resources are increasingly over-utilized.

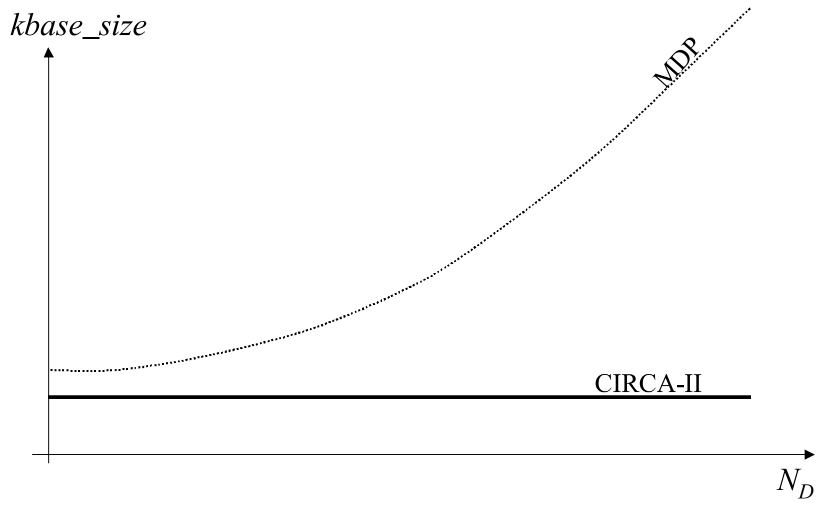


Figure 4-19: Knowledge Base Size vs. N_D for MDP and CIRCA-II planners.

Recall that in Chapter II we described the Markov Decision Process (MDP) and argued that to develop general real-time control plans for guaranteed failure-avoidance, we require a "k-level" MDP model in which the state transition matrix has worst-case size $(N_A\, xN_D)$x N_S^k x N_S^k, where N_A is the number of unique actions, N_D is the number of different deadlines assigned to each action, N_S is the number of modelable states and is exponential *(v^f)* in the number of unique state features *(f)* and values *(v),* and k is the number of previous stages that must be effectively "remembered" in the current state. CIRCA-II produces a sufficient but not optimal control plan, and contains only an approximate conditional probability model. However, CIRCA-II completely avoids the MDP complexity due to N_D by explicitly computing deadlines for its set of N_A actions and incorporating their effects directly into the online conditional probability computations described in this chapter.[40]

Figure 4-19 illustrates the knowledge base savings in CIRCA-II over the *unsimplified* MDP (i.e., MDP with no state aggregation, etc. used for efficiency gains) as N_D grows large, where the initial offset in *kbase_size* is due to CIRCA-II's use of abstract state transitions instead of a full set of conditional probabilities for all states. Large N_D will also complicate MDP plan generation because it must search through the full set of *(N_D*N_A)* actions for each state for each iteration when building an optimal policy. In cases with limited backtracking, CIRCA-II planning will have a distinct efficiency edge over MDP planning. However, in the worst-case, both planners will be slow, MDP because of the large set of action deadlines, and CIRCA-II because of its necessity to exhaustively backtrack through all possible combinations of actions and states to find a schedulable plan.

CIRCA-II directly addresses the difficulty of "remembering a k-stage history" in the MDP formulation via the use of time-dependent transition P_{rate} functions and the propagation of dependent temporal transitions effects throughout the expanded state-

[40] In fact, since CIRCA-II can assign any numerical value as an action deadline, the MDP would theoretically require infinite N_D to fully-emulate the flexibility of CIRCA-II when assigning deadlines. Our experiments typically have required fewer than ten N_D values for each action over the duration of a mission, but we would not have been able to easily predict the exact set of N_D values required until after the mission plans were developed.

space.[41] Additionally, since we use STRIPS-like abstract preconditions and postconditions when specifying a domain model, a CIRCA-II knowledge base will generally contain far fewer entries than even the total number of modelable states N_S. The expected divergence in CIRCA-II and MDP knowledge base size as the modelable state-space N_S increases is illustrated in Figure 4-20 for different k. Again, we again cannot readily generate execution time or memory usage trends for MDP vs. CIRCA-II planner for varying N_S or k. However, for large k and N_D, the CIRCA-II planner certainly has a "good" chance of out-performing the MDP planner unless a worst-case backtracking scenario is encountered.

The evaluation presented in this section is strictly qualitative and based on complexity and/or expected results. The intent of this section is to motivate our use of the specific CIRCA-II techniques for complex problem domains that require tradeoffs due to insufficient plan-execution resource capacity. We have biased our MDP formulation to produce the control plans required for CIRCA-II and compared it to a planner designed specifically to produce control plans as we have defined them. Thus, we emphasize that this section is not attempting to diminish the importance of MDP-based planners, but rather to argue that there is cause to develop an alternate methodology such as that presented in this chapter.

Even with our tailored planning formulation, we are only able to generate sufficient plans that will maintain system safety, whereas an MDP planner will be able to produce *optimal* plans. We will continue to analyze the tradeoff between MDP and state-space planners for CIRCA-II, particularly as we transition to the realm of time-

[41] Dependent temporal transition computations cover the specific history information we require for CIRCA-II state probability computations given our P_{rate} functions. As an analogy to the k-level MDP, each *dtt* in our state-space will be continosly active through a sequence of <u>at most</u> k states.

constrained dynamic planning and are able to compare a future anytime [15] CIRCA-II planner (see Appendix D) with a bounded-optimal MDP [62].

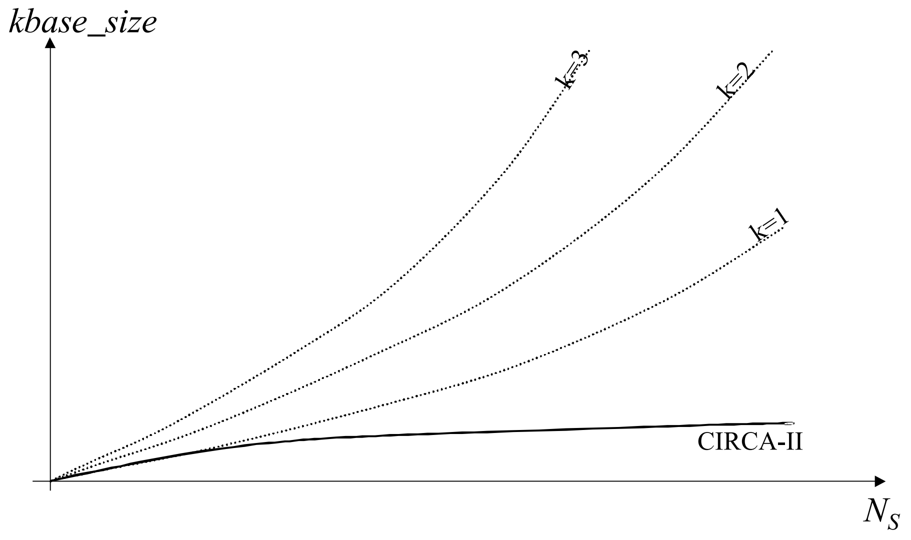


Figure 4-20: Knowledge Base Size vs. # of Modelable States.

CHAPTER V
DETECTING AND REACTING TO ANOMALOUS EVENTS

We require a system that is capable of safely operating with imprecise knowledge and incomplete plans, resulting in the possibility of reaching states during plan execution that were not expanded during planning. If ignored, the consequences of such "unhandled" states may be disastrous, particularly if these states are dangerous and no reaction fortuitously guides the system back to a safe path.

As defined in Chapter I, our control plans are a set of actions with minimized preconditions (i.e., potentially costly feature tests) that match multiple world states. Each plan is specified as a cyclic task schedule to guarantee hard real-time response in dangerous world states. A more traditional plan specifies a list (database) of states and corresponding actions to execute in each state. For a probabilistic or nondeterministic state-space, the traditional plan (policy) requires complete state feature sensing at each plan step and a search for the appropriate state-action combination, a procedure which cannot guarantee timely action retrieval and execution thus cannot guarantee safety in our hard real-time environment.

A system using our definition of a control plan cannot simply “know” when it has deviated from the set of states for which an executing plan is valid. Instead, a plan must contain explicit directives for determining when such a deviation has occurred. We define a *handled state* as a situation which has been expanded by the planner and both lies along a goal path and is safe (i.e., no unpreempted *ttfs* exit from the state). All other states are classified as *unhandled*. A control plan reacts to all handled states appropriately, thus no further intervention is required when such situations are encountered. In this section, we identify classes of unhandled world states we argue are the most important to detect, describe the algorithms used by the CIRCA-II planner to build detection TAPs for these states, and then present an algorithm for maintaining safety and responding appropriately when an unhandled state is actually observed.

World State Classification

We approach the problem of detecting important unhandled states by first developing a classification of all world states. Then, we can gain efficiency by exclusively enumerating and building reaction mechanisms for unhandled (or unplanned-for) states that we classify as "important". Figure 5-1 shows the relationship between subclasses of possible world states. Modeled states have distinguishing features and values represented in the planner's knowledge base. Because the planner cannot consider unmodeled states without a feature discovery algorithm, unmodeled states are beyond the scope of this paper. "Planned-for" states are those the planner has expanded. This set is divided into two parts: "handled" states from which failure is assured to be avoided and from which the goal can be reached, and "deadend" states from which failure is avoided but from which the goal cannot be reached using the current plan.

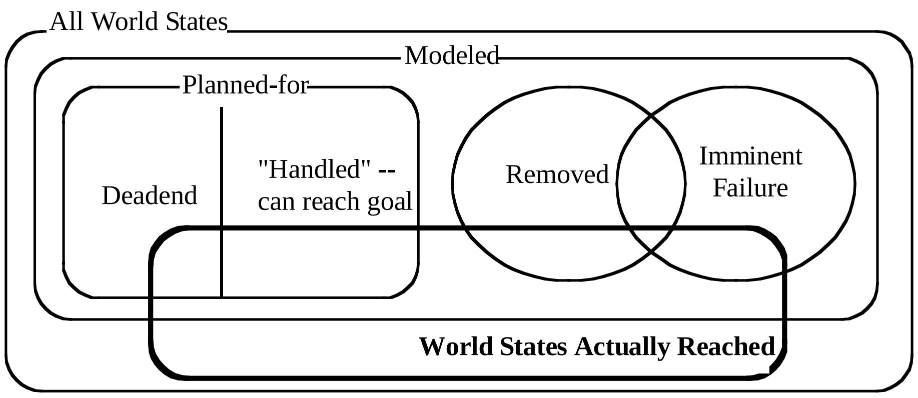


Figure 5-1: World State Classification Diagram.

A variety of other states are modelable by the planner. Such states include those identified as reachable, but “removed” because attending to them along with the “planned-for” states exceeds system capabilities. Other modeled states include those that indicate “imminent failure;” if the system enters these states, it is likely to fail shortly thereafter. Note that some states might be both “removed” and “imminent-failure”, as illustrated in Figure 5-1. Finally, some modeled states might not fall into any of these categories, such as the states the planner considered unreachable but that are not necessarily dangerous. As illustrated by the boldly outlined region in Figure 5-1, states actually reached may include any subclass. To assure safety, the set should only have elements in the “planned-for” region. When the set has elements outside this region, safety and performance depend on classifying an unplanned-for state if and when it is entered and responding appropriately. For this reason, we provide more detailed definitions of the most important classes.

A "deadend" state (DS) results when a transition path leads from an initial state to a state that cannot reach the goal, as shown in Figure 5-2. The deadend state is safe because there is no transition to failure. However, the planner has not selected an action that leads from this state via any path to the goal. Deadend states produced because no action can lead to a goal are called "by-necessity", as when an arriving aircraft cannot reach its designated airport gate because it is occupied by another plane. Conversely, those deadend states produced because the planner simply did not choose an action leading to the goal are called "by-choice”, often created in order to decrease plan complexity or to avoid the potential for future system failures. For example, an aircraft with a goal destination airport might build a plan that invokes actions to land at the nearest airport upon approach to any inclement weather, effectively producing deadend states whenever a “bad weather ahead” transition occurs. These states may be deadend “by-choice” because a more complex plan might have been able to invoke actions that allow the aircraft to safely go around the bad weather, but the planner chose simplicity over completeness since system safety was not at stake.

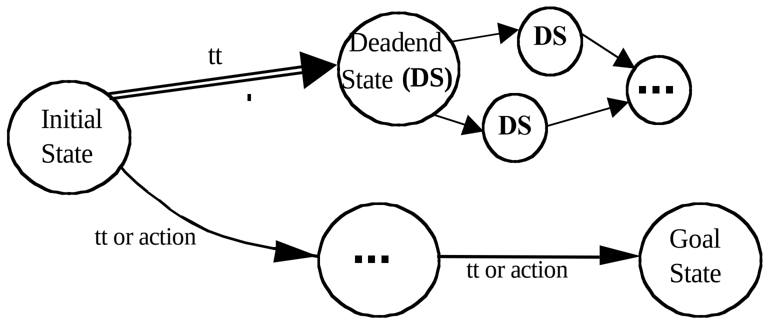


Figure 5-2: "Deadend State" Illustration.

A “removed” state results from the planner’s inability to guarantee that the system will avoid failure. A planner that generates real-time control plans needs to backtrack whenever scheduling fails. When backtracking, the planner may select different actions so long as failure is still avoided. However, even after exhaustive backtracking, a planner may fail to find actions that meet all objectives while still being schedulable. One option is ignoring some reachable states, thus not planning actions for them. A control plan so constructed cannot claim to be foolproof. However, for real-time control applications, it may be more important to make timing guarantees under assumptions that exceptional cases will not occur than to make no guarantees about a more inclusive set of cases. Our heuristic for pruning states is to overlook the most unlikely states. A "removed" state set is created when the planner has purposely removed the set of lowest probability states, as illustrated in Figure 5-3. In the first planner iteration, all states with nonzero probability are considered, as depicted by the "Before Pruning" illustration. Here the planner must consider a low probability transition leading to a state which transitions to failure, and must guarantee a preemptive action to avoid failure.

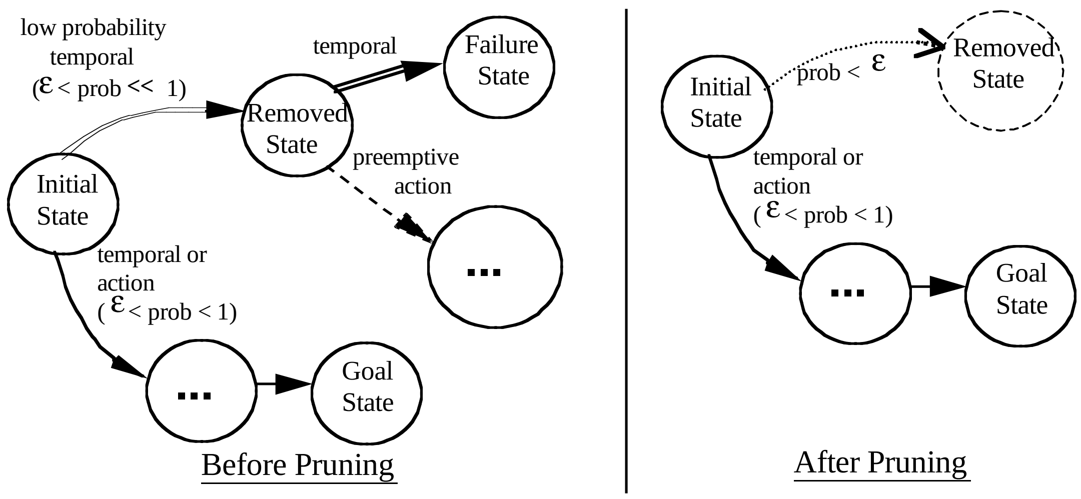


Figure 5-3: "Removed State" Illustration.

Suppose the scheduler fails. The planner will backtrack and build a new plan without low-probability states. The resulting state diagram -- "After Pruning" -- is shown in Figure 5-3. All states downstream of the low probability transition in the new plan are no longer expanded. The preemptive action is no longer required, giving the scheduler a better chance of success. A flight simulation example with removed states is shown in Chapter VII, illustrating why removal of these states was necessary and how the system successfully detected and reacted to these improbable situations.

During plan development, all temporal transitions to failure (*ttfs*) from reachable states are preempted by guaranteed actions. If preemption is not possible, the planner fails. However, the planner does not worry about *ttfs* from any states it considers unreachable from the initial state set. The set of all modelable states considered unreachable that also lead via a modeled state transition to failure are labeled "imminent-failure".[42] Actually reaching one of the recognizable imminent-failure states indicates

[42]It is also possible that states that are unmodelable could lead directly to failure with a known transition, or that modelable states could lead directly to failure with transitions not known to the planner, or that states that are not modelable could lead directly to failure with an unknown transition. We exclude these cases from the “imminent-failure” set because the planner is incapable of classifying them in this way.

either that the planner's knowledge base is incomplete or incorrect (i.e., it failed to model a possible sequence of states), or that the planner chose to ignore this state to allow other guarantees.

Figure 5-4 shows a diagram of a reachable state set along with an isolated state (labeled "Imminent-failure") leading via one temporal transition to failure. This state has no incoming transitions from a reachable state, so the planner will not consider it during state expansion. However, if this state is reached, the system may soon fail. The imminent-failure unhandled states are important to detect because avoiding system failure is usually a primary system goal. Consider an aircraft that "trusts" air traffic control to maintain traffic separation. During flight through controlled airspace, such an aircraft would have no model of a transition to a state in which another aircraft is on a collision-course. However, a mid-air collision, if it does occur, usually leads to catastrophic failure. During controlled-airspace flight, we model the "collision-course traffic" scenario as an imminent-failure state for the aircraft that implicitly trusts air traffic control, provided the knowledge base includes a *ttf* that describes the failure when two aircraft collide.

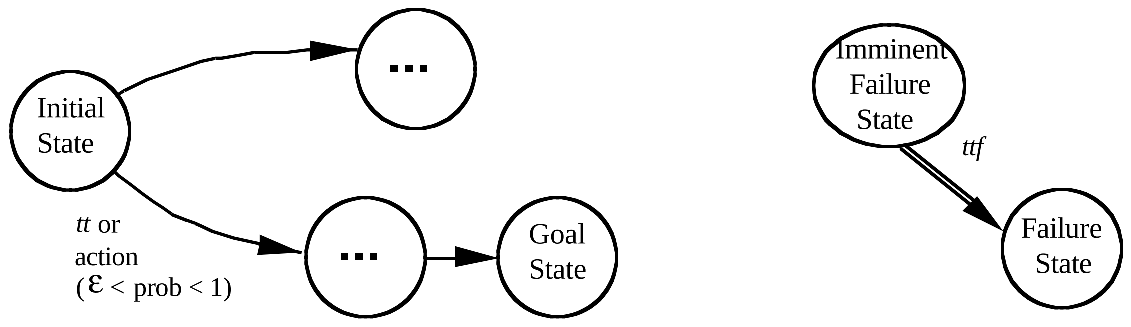


Figure 5-4: "Imminent-failure State" Illustration.

Detecting Unhandled States

As discussed above, a planner cannot be expected to somehow just “know” when it has deviated from plans---it must explicitly plan actions and allocate resources to detect such deviations.[43] We have identified classes of "unplanned-for" states that we consider important to detect, and in this section we describe the methods we use to actually detect these states should they occur during plan execution.

In our CIRCA-II implementation, after the planner builds its normal plan, it builds special TAPs to detect deadend, removed, and imminent-failure states. Other unhandled states, such as those “modeled” but outside “planned-for”, “removed”, and “imminent-failure” regions in Figure 5-1, are not detected by CIRCA-II. If it reaches an unhandled state that is not detected by CIRCA-II, the system may transition back to a planned-for state (where the original plan executes properly), transition to an imminent-failure state (where CIRCA-II will detect the state and react), or simply remain safe forever without reaching the task-level mission goals.

We have developed algorithms to build detection tests for deadend, removed, and imminent-failure states. CIRCA-II TAPs *could* include tests for every set of features in that unhandled state list (e.g., each deadend state), but these tests would be repeated frequently during plan execution and may be time-consuming, as in PRS [32], where context checking could involve a large, non-minimal number of tests, including updates from sensors. In CIRCA-II, once an unhandled state list is completed, the planner calls ID3 [57] using the information-gain splitting heuristic with all of that class of unhandled states as positive examples and all “planned-for” states not also in the unhandled state class as negative examples. ID3 returns what it considers a minimal test set which is then

[43] This premise is consistent with both CIRCA and CIRCA-II control plan designs in which each state must be classified to see if a particular TAP's action applies. CIRCA-II is distinct from CIRCA in that CIRCA-II takes the "extra step" to classify and react to states outside the planned-for set.

used to detect that unhandled state class. During plan execution, when any of the unhandled state detection TAP tests are satisfied, the plan executor notifies the plan cache (and subsequently the Planning Subsystem) of this deviation. Below, we discuss how CIRCA-II builds lists of the different unhandled state classes.

Although they do not result in system failure, deadend states have no temporal transitions or planned actions that ever lead to a goal state, thus are important to identify. To build a list of these states, CIRCA-II searches the transition links from each state to the goal state(s). If no goal is found, that state is labeled "deadend" while the reachable states along a goal path are labeled "non-deadend". Deadend states are the positive ID3 examples and non-deadend states are the negative examples.

Whenever the CIRCA-II Planning Subsystem backtracks due to scheduling difficulties, low-probability states are removed from consideration in the current plan based on a minimum probability threshold ($P_{removed}$) below which states are considered unreachable by the planner. This threshold is gradually incremented until a schedulable “nominal” plan based on the most likely set of states is produced.[44] The set of “reachable” (planned-for) states is defined as those expanded during development of this nominal plan. The remaining states on the state stack that have *not* been expanded (i.e., because their probability is less than $P_{removed}$) are then set as "initial states", and the planner executes its state expansion routine on these states presuming this plan’s planned actions are executed. All states expanded, including the initial states, are included in the list of “removed” states because they are possible to reach from the nominal plan, even though the likelihood of reaching such states may be minimal. To build the removed

[44] For some domains, it may be desirable to assign a maximum value for $P_{removed}$. If no schedulable plan can be found without exceeding this limit, then the CIRCA-II planner will fail.

state detection tests, ID3 is called with this “removed” state list as positive examples and all states expanded during nominal plan creation as negative examples.[45]

While the planner should look for deadend and removed states because they are more likely to occur than other unhandled states (i.e., the planner knows they could arise), likelihood is not the only criterion for allocating resources to detection. No matter how unlikely, detecting imminent-failure states is important because of the potentially catastrophic consequences of being in such states. When building imminent-failure state sets, we assume the modeled set of temporal transitions to failure (*ttfs*) is complete and correct, even though reaching such a state implies at least one other transition is not accurately modeled. The planner begins with a list of all precondition feature sets from *ttfs*. This list is expanded to fully enumerate all possible states that would match these preconditions. Any reachable states are removed from this list. Imminent-failure detection TAP tests are then built with this list as ID3 positive examples and the reachable states as negative examples. Note that a complete list of fully-instantiated states to detect can be quite large. We continue to search for alternatives or approximations to ID3 that will provide sufficient accuracy with improved computational efficiency, although planning efficiency will not be of paramount importance to CIRCA-II until we impose real-time bounds on our planning processes.

---

[45] A simpler version currently implemented for removed state generation is to consider all states remaining on the state list with probability below $P_{removed}$ as the set of removed states. This is the first-level group of removed states. However, they may not be sufficient for detection of all deviations into "removed state regions" since such states may be transient and thus not still be present when the corresponding detection TAP executes.

Real-time Reaction to "Unplanned-for" States

For real-time operation, an automated system must guarantee both a timely and accurate response. CIRCA-II utilizes explicitly-scheduled control plans to guarantee that the system will be safe so long as the environment remains within the set of planned-for states (see Figure 5-1). In this section, we describe a method to improve CIRCA-II's ability to also remain safe in the important unplanned-for state classes we have defined above.

Because accurate planning is generally an NP-complete problem, online planning must be invoked only when time is available, with offline deliberations utilized to develop reactions for the most time-critical situations and online planning to react to other, “safer” states. This combination of offline and online planning is desirable because it is generally infeasible to create a universal plan set [63] in complex problem domains (using strictly offline deliberation), as discussed in [23], but, conversely, it is also impossible to guarantee timely online planning responses when the response must occur very quickly. To directly address the tradeoff between [slow] dynamic planning and a large pre-defined plan database, we require a planner that explicitly computes available deliberation time based on the notion of failure avoidance. Next, the planner uses these time-to-failure computations to identify situations in which pre-computed plans are mandated for failure avoidance, versus situations in which dynamic planning can occur with sufficient speed to avoid any far-term failures that might occur.

Figure 5-5 illustrates the idealistic “plan-space” concept we adopt as our failure avoidance model. In Figure 5-5, plan execution begins in “Planned-for States 1”, and in best-case situations, execution will remain within that block using reactions from the executing plan. However, we account for cases in which some state (temporal) transition (*tt*) leads away from that block, either via a model inaccuracy or plan incompleteness. When such a deviation occurs, we must identify it and react in sufficient time to avoid catastrophic failure, modeled in the figure by a state transition to failure (*ttf*). In this paper, we describe a method for detecting such "unhandled" states, with primary attention paid to those requiring time-critical responses.

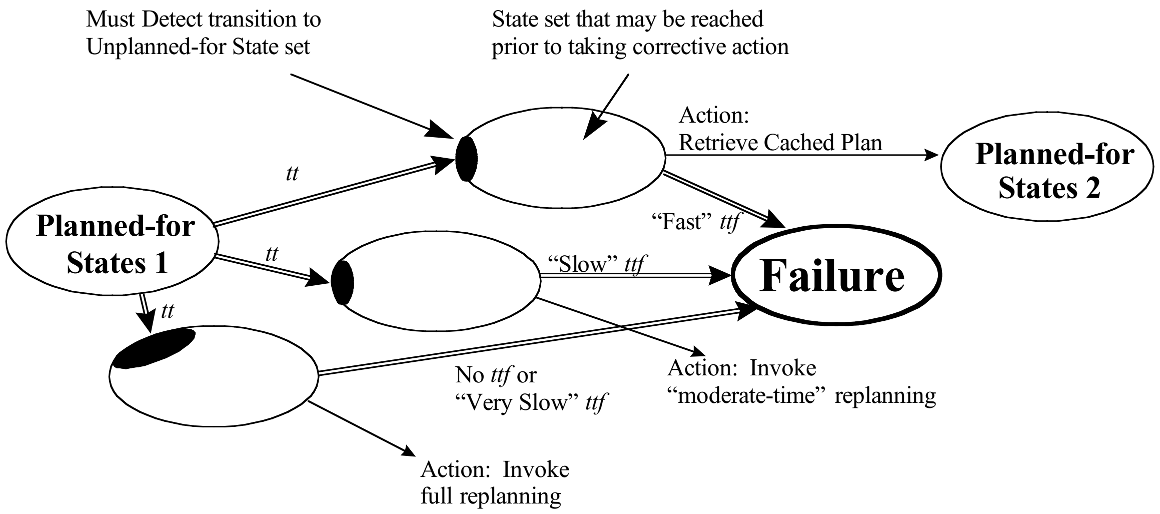

Must Detect transition to Unplanned-for State set
State set that may be reached prior to taking corrective action
Action: Retrieve Cached Plan
Planned-for States 2
tt
"Fast" ttf
Planned-for States 1
tt
"Slow" ttf
Failure
tt
No ttf or "Very Slow" ttf
Action: Invoke "moderate-time" replanning
Action: Invoke full replanning

Figure 5-5. Plan-space Transitions based on Time to Failure.

If a *"fast ttf"* matches an unhandled state, a pre-computed plan must be quickly executed before failure can occur, since insufficient time exists for online deliberation. Otherwise, an appropriate planning algorithm may be selected based on how much time remains before failure can occur.[46] For example, as shown in Figure 5-5, the "moderate time" planning algorithm might be a case-based technique [28] for which worst-case execution time is a function of case database size, or an anytime planner [15],[74],[75] which includes a careful definition of worst-case execution time for development of a "minimum quality" plan to avoid failure. The "full replanning" algorithm may be a state-

[46] In the most general case, more than two planning algorithms might be available. Adopting a design-to-time approach [21], additional classifications of ttf "speed" would then be identified such that each planner would be invoked based on ttf "speed" from the identified unhandled state.

based planner with difficult-to-predict execution properties, often required for complete and precise plan development. We assume planning accuracy is a function of deliberation time, so that the “full replanning” algorithm will produce the most detailed (best) response set.

We consider the Figure 5-5 "plan-space" concept with any number of intermediate planning levels to be idealistic because, for each level, one must carefully define the utility of selecting that level for responding to the next state versus the next "higher" or "lower" planning level. With many planning algorithms, it is challenging to clearly specify result accuracy and response timeliness without actually performing the planning operation. In this thesis, we present a two-level "plan-space" design and describe how it specifically fits into CIRCA-II.

Figure 5-6, illustrates the plan-space diagram utilized in CIRCA-II. We incorporate two distinct methods for handling unplanned-for states: plan retrieval from the cache and replanning. As discussed previously, CIRCA-II contains an “unbounded” state-space planner and a plan cache that must have “bounded” retrieval times. CIRCA-II dynamic replanning should only be required from unhandled states that cannot lead to failure (e.g., deadend states), while the plan cache should contain a failure-avoidance plan for all unhandled states (e.g., “dangerous” removed and imminent-failure) with *ttfs* that occur quickly. As depicted in Figure 5-6, the contingency plans may not re-direct the system to the “planned-for” state set, because they focus exclusively on failure-avoidance. Instead, they redirect the system to states with a longer time before failure can occur, effectively “buying time” for the CIRCA-II planner to construct a new goal-achievement plan.

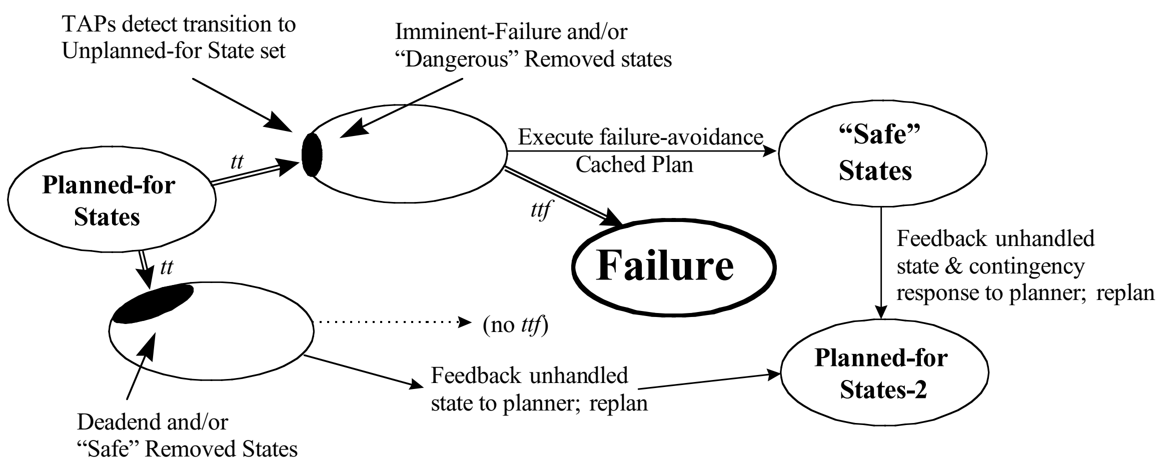

TAPs detect transition to Unplanned-for State set
Imminent-Failure and/or "Dangerous" Removed states
Planned-for States
tt
Execute failure-avoidance Cached Plan
"Safe" States
ttf
Failure
Feedback unhandled state & contingency response to planner; replan
tt
(no ttf)
Feedback unhandled state to planner; replan
Planned-for States-2
Deadend and/or "Safe" Removed States

Figure 5-6: Plan-space Transitions in CIRCA-II.

As in the original CIRCA, CIRCA-II control plans are explicitly scheduled such that failure-avoidance is guaranteed while in any set of planned-for "probable" states. The remaining challenge is to meet all hard real-time deadlines required for successfully responding to each unhandled state with a contingency plan. To maximize plan retrieval efficiency, the plan cache is empty upon system startup[47], then offline planning populates the two cache partitions, nominal and contingency, and assigns each contingency plan a "matching" nominal plan. Therefore, when an unhandled state is encountered, only the set of contingency plans associated with the currently-executing nominal plan must be searched for a match with the unhandled state.

Ideally, the CIRCA-II planner will have built a set of contingency plans that can be retrieved quickly enough to make hard real-time guarantees of failure-avoidance in all time-critical unhandled states. However, in practice, some unhandled states may have sufficiently fast *ttfs* that it is impossible to assuredly retrieve a contingency plan in time,

---

[47] In future work we plan to study tradeoffs associated with "remembering" or "learning" contingency plans that are frequently used to minimize the overhead during startup currently required to populate the cache from scratch.

particularly when a large set of unhandled states require a large group of contingency plans. For each nominal plan, we can assess CIRCA-II's ability to make real-time failure-avoidance guarantees for unhandled states. First, the worst-case contingency plan retrieval time (t_r) may be computed from Equation 5-1, where n is the number of contingency plans associated with the executing nominal plan, and t_{max} is the maximum time required to test for an unhandled state match in any of these contingency plans.

$$t_r = n * t_{\max} \qquad (5\text{-}1)$$

For failure-avoidance to be guaranteed in an unhandled state, Equation 5-2 must be satisfied, where $min\Delta(s_u)$ is the minimum delay before any ttf from unhandled state s_u can occur ($min\Delta$ definition for a tt was described earlier in Equation 4-11), t_d is the worst-case time between the occurrence and detection of state s_u (equal to the execution period of the nominal plan's "unhandled state" detection TAP for that state class), and t_{over} is the near-constant overhead time required to actually start up the new plan once retrieved.

$$min\Delta(s_u)\ ?\ (t_r + t_d + t_{over}) \qquad (5\text{-}2)$$

Using Equations 5-1 and 5-2, we are able to identify the set of time-critical unhandled states that achieve guaranteed failure avoidance. In many cases, CIRCA-II may make absolute failure-avoidance guarantees for all states, even for "improbable" states handled via contingency plans. However, time-critical situations may exist in which Equation 5-2 does not hold for all the unhandled states. In these worst-case scenarios, CIRCA-II will perform analogously to an overloaded real-time system -- it will guarantee a subset of all failure-avoidance reactions but will only be able to achieve best-effort response for the others.

We have assigned a preference in the CIRCA-II planner for building contingency plans for “dangerous removed” states before those for “imminent-failure” states, since both are critical for failure-avoidance but removed states have a small but non-zero probability, whereas imminent-failure states will occur only if the model is inaccurate. With contingency plans retrieved in first-in-first-out (FIFO) order, the most “likely” set of improbable states will have the minimal retrieval times (i.e., their matching contingency plans will be searched first), thus the plan cache will be able to meet plan retrieval deadlines more often when the model is accurate.

Figure 5-7 shows a qualitative plot of expected CIRCA and CIRCA-II performance as a function of [inverse] plan-execution resource capacity *c*. Previously (Figure 4-12), we illustrated how CIRCA-II achieves flexibility in trading off the probability of mission success (i.e., safety) for schedulability when resources would otherwise be over-utilized. In Figure 5-7, we add a new curve (labeled "CIRCA-II w/ cache") that illustrates how the detection of unhandled states and incorporation of the CIRCA-II plan cache improves overall chances for success, even when the model is not precise (so imminent-failure states may be reached) and resources are limited (so states may require removal). As shown in the Figure, safety is guaranteed with 100% likelihood *($P_{success}$=100%)* as with CIRCA until resource capacity requires incrementing $P_{removed}$ about 0. Then, performance will drop off until no actions can be scheduled to execute in a timely fashion, giving a near-zero chance of success regardless of how many plans are cached. The distinction between CIRCA-II with and without the real-time cache is simply that, in the region where the resource set is *marginally over-utilized*, the plan cache enables CIRCA-II to more effectively utilize its resources for failure-avoidance by moving through a hierarchy of real-time schedules that guide the system safely through its environment.

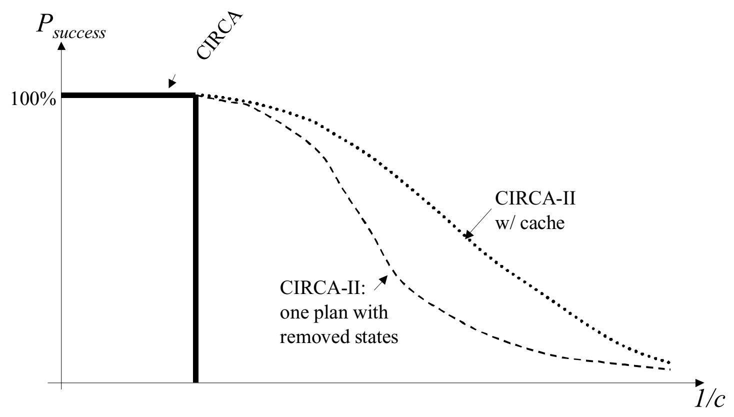


Figure 5-7: Success Probability vs. Resource Capacity: With and Without Plan Cache.

CHAPTER VI
PLANNER-SCHEDULER NEGOTIATION

Planning for real-time applications involves decisions not only about what actions to take in what states, but also about how to realize those actions within hard real-time deadlines given the inherent resource limitations of an execution platform. Determining how to arrange planned actions in a sequence such that timely execution is guaranteed within constraints is a manifestation of the scheduling problem. We adopt a modular approach that couples separate planning and scheduling components into the CIRCA-II architecture, so that the planner and scheduler can separately apply their expertise to ultimately build an appropriate plan that will execute with hard real-time guarantees on the plan execution platform.

The planner is an expert at determining which tasks must be performed subject to which constraints to solve the global problem at hand, while the scheduler is an expert at manipulating the tasks into a specific order such that constraints are not violated. Ideally, one would like the scheduler to know only how to manipulate tasks into a cyclical sequence which does not violate constraints, while a planner knows about the global problem at hand and the tasks required to solve the problem, but not the details of how to organize the tasks into a schedule. For communication between planner and scheduler, however, the two must share some knowledge. How much knowledge should be shared and how to represent this knowledge is not clear. This problem generally requires iteration between developing alternative plans and evaluating the schedulability of those plans until an executable plan that maximally accomplishes goals is found.

In this chapter, we describe our efforts to identify and utilize a minimal but expressive shared knowledge representation for coupled planner and scheduler agents, and describe how this information supports the iterative formation of real-time guaranteed control plans. We have approached this work in several stages. First, we look at the specific case of a single-processor plan execution platform and describe how the CIRCA-II scheduler computes and feeds back resource usage information to help guide replanning efforts when plan scheduling fails. Next, we admit a general multi-resource plan execution platform and describe how generic allocation and scheduling algorithms may provide "bottleneck task" feedback for guiding planner backtracking efforts when scheduling fails. A major advantage of a multi-resource execution platform is the ability to introduce fault-tolerance. We discuss a limited fault-tolerance methodology for CIRCA-II, based on the multi-resource allocation and scheduling algorithms and the CIRCA-II plan dispatcher. Finally, we venture into a discussion of how Quality-of-Service (QoS) negotiation may be incorporated into CIRCA-II such that both the planner and scheduler can make tradeoffs when scheduling the "ideal" plan is impossible given the limited execution resources.

Scheduler-to-Planner Feedback[48]

The single-processor CIRCA-II scheduler is based on a *non-preemptive separation-constrained* method of scheduling described in [52]. The scheduler simulates the execution of a dynamic scheduler by maintaining a time counter and iteratively

[48] This work was done cooperatively with C. B. McVey. Further details of the CIRCA-II Schedule Manager, including the computation of under-utilization parameters and a more detailed specification of the feedback message structure, are provided in [47].

incrementing it as TAPs are chosen for execution. At each iteration, the TAP with the shortest slack time is chosen to be executed. TAP slack time $(t_{slack}(TAP_i))$ is defined in Equation 6-1, where $t_{sep}(TAP_i)$ is the TAP's separation constraint specified by the planner, t_{cur} is the current time, and $t_{last}(TAP_i)$ is the time TAP_i was last chosen to execute.

$$t_{slack}(TAP_i) = t_{sep}(TAP_i) - (t_{cur} - t_{last}(TAP_i)) \qquad (6\text{-}1)$$

If any other TAP (TAP_j) has sufficiently small worst-case execution time $(wcet(TAP_j))$ to fit within the slack time of the originally-chosen TAP $(wcet(TAP_j) < t_{slack}(TAP_i))$, it will be selected for placement in the schedule instead. If the slack time of any TAP is less than zero at any point, the TAP's deadline is violated and scheduling fails. After all TAPs are present in the schedule, the scheduler continues its simulation until a valid periodic subsequence containing all TAPs is extracted as the final schedule.

In this section, we first describe the Schedule Manager which was added to the original CIRCA scheduler to construct and direct message-passing between planner and scheduler. Then we provide an example that illustrates the use of this feedback.

Schedule Manager

A scheduler capable of providing meaningful feedback to a planner must have authority to manipulate and retry scheduling the requests it receives from the planner. Given this capability, the scheduler can use the difference between a satisfiable request and over-constrained request to provide more accurate feedback to the planner. We have augmented the original CIRCA scheduler with a rule-based system (the "Schedule Manager") which directs the processing of all scheduling requests from the planner. Depending upon the request, this Manager may perform a variety of actions: schedule a request, modify some constraints in a request, modify parameters which govern behavior of the core separation-constrained scheduling algorithm, calculate appropriate feedback for the planner, and transmit a valid schedule or feedback.

A high-level summary of Schedule Manager algorithm is shown in Figure 6-1. Upon receiving a schedule request from the planner, the manager first checks that overall processor utilization *U* is less than one and tests for task-pair conflicts (i.e., whether every combination of two guaranteed tasks will fail to fit together on the single processor given their separation constraints and *wcets*). If both of these tests "pass", the Manager calls the scheduler. Otherwise, it concludes the scheduler will have no chance at complete success with the current plan and constructs feedback for the planner.

The primary feedback from Schedule Manager to planner is a suggested probability threshold $(P_{removed})$ below which "unlikely" states are to be ignored. This $P_{removed}$ recommendation is made based on a heuristic-guided binary search between the minimum, maximum, and current threshold that have been used during development of this plan. When the planner adopts an increased probability threshold, the state-space search is pruned, effectively generating the "removed" states described previously in Chapter V. This pruning ultimately results in increased TAP separation constraints and/or the removal (or replacement) of some TAPs from the scheduling request.

Scheduler Feedback Example

We have incorporated the Schedule Manager into CIRCA-II, and here present a simple flight domain example to illustrate the utility of scheduler-to-planner feedback. Figure 6-2 shows the state-space from an automated flight simulation test run to test the CIRCA-II Schedule Manager algorithms. Note that, for state diagram conciseness, the "tornado" temporal transition, which is very unlikely but matches every state while the aircraft is in flight, leads to a "generalized" state with (Tornado = T), which must be countered with the "avoid-tornado" action that leads back to the "pre-tornado" state.

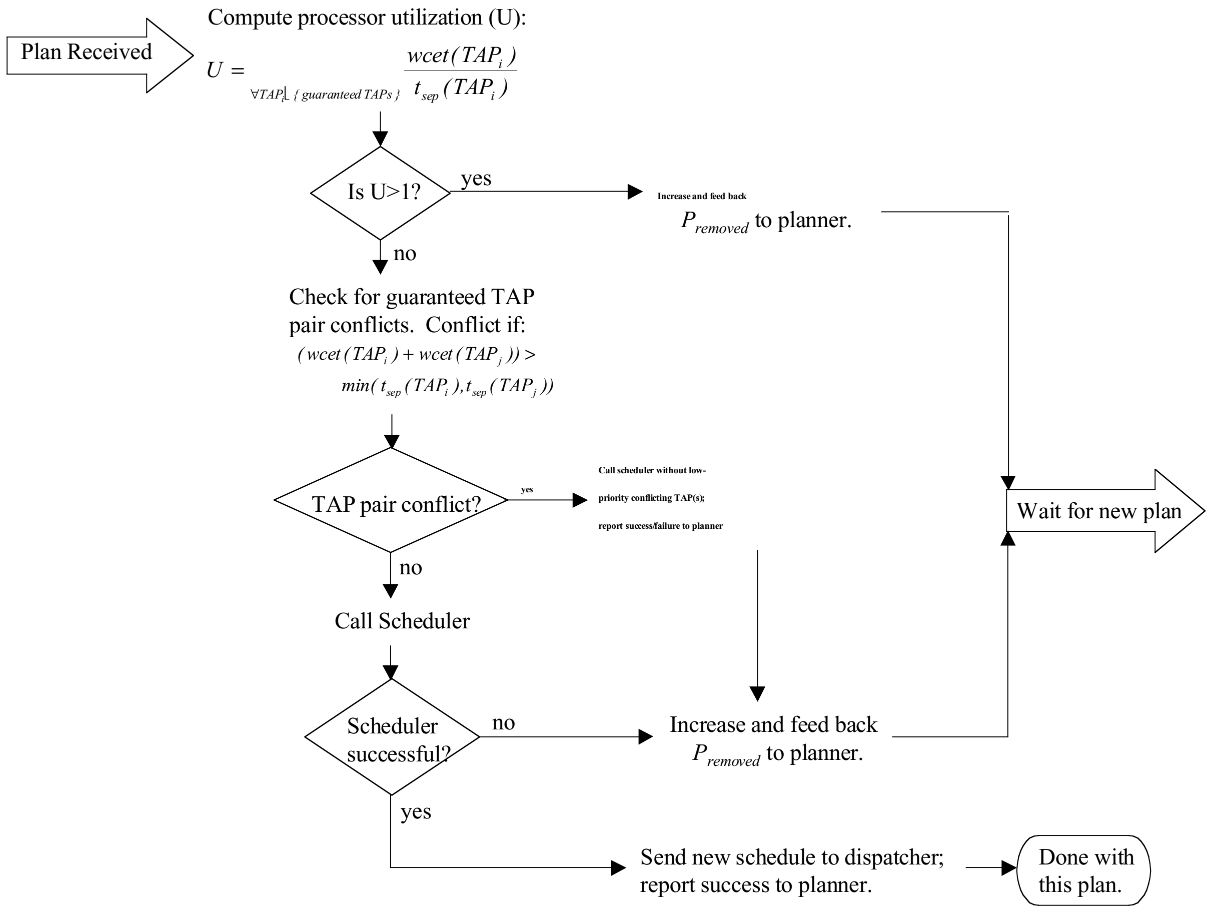

Plan Received
Compute processor utilization (U):
$U = \sum_{\forall TAP_i \in \{guaranteed\ TAPs\}} \frac{wcet(TAP_i)}{t_{sep}(TAP_i)}$
Is U>1?
yes
Increase and feed back
$P_{removed}$ to planner.
no
Check for guaranteed TAP pair conflicts. Conflict if:
$(wcet(TAP_i) + wcet(TAP_j)) > min(t_{sep}(TAP_i), t_{sep}(TAP_j))$
TAP pair conflict?
yes
Call scheduler without low-priority conflicting TAP(s); report success/failure to planner
no
Call Scheduler
Scheduler successful?
no
Increase and feed back $P_{removed}$ to planner.
yes
Send new schedule to dispatcher; report success to planner.
Done with this plan.
Wait for new plan

Figure 6-1: CIRCA-II Schedule Manager Algorithm.

Table 6-1: Required TAP set (+ *if-time-server*) for "Traffic Avoidance" Plan.

TAP #	Name	*wcet*	t_{sep}	Probability	Priority
0	*if-time-server*	3550	N/A	N/A	N/A
1	*climb*	2150	45000	1.0	41
2	*avoid-tornado*	4150	9000	0.057	15
3	*avoid-traffic*	2150	20000	0.943	47
4	*course-correct*	5325	90000	0.9	41
5	*resume-heading*	2150	45000	0.89	38

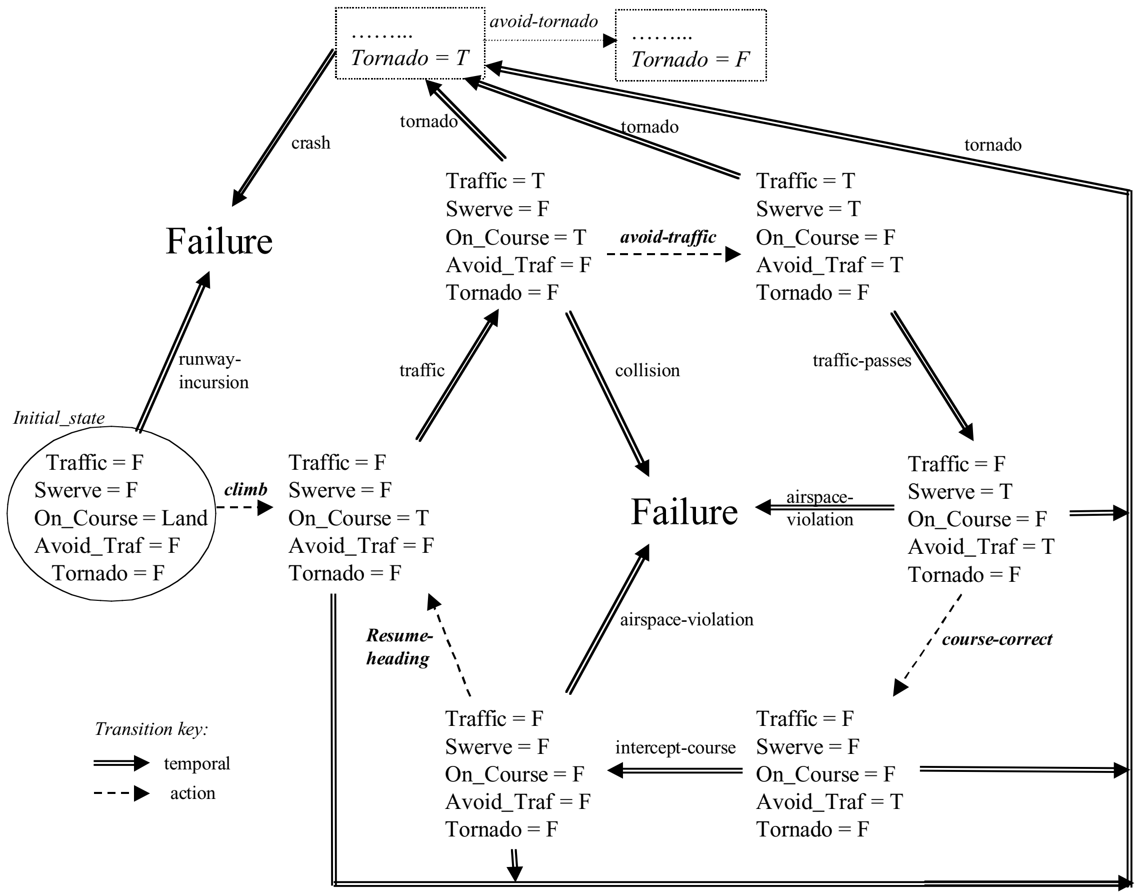


Figure 6-2: CIRCA-II "Traffic-Avoidance" Excerpt from Automated Flight State-Space.

The set of "guaranteed" actions required to avoid *ttfs* are listed in Table 6-1. We present for this example the output from one possible planner-scheduler iteration cycle from this state-space. Following the Figure 6-1 algorithm, the schedule manager first determines that U<1, so it looks for TAP pair conflicts. Unfortunately, TAP #2 conflicts with Tap #4 since the sum of their wcets is greater than TAP #2's separation constraint *(t_{sep})*. At this point, the scheduler computes and returns $P_{removed}$ to the planner. To compute $P_{removed}$, the scheduler performs a binary search to find the maximal set of guaranteed TAPs that can be scheduled, then sets $P_{removed}$ to the highest-probability value for the TAPs that did not fit into this set (see [47] for further details). For our example, $P_{removed}$ is set to 0.057 since a successful schedule could be constructed with all other

TAPs. Upon replanning, TAP #2 disappears because all "Tornado=T" states fall below this probability threshold. Thus the scheduler is successful on the next pass, constructing and downloading the TAP schedule *{3, 1, 0, 4, 0, 3, 5, 0}*, where schedule numbers correspond to TAP numbers and the "if-time server" (TAP #0) has been inserted as frequently as possible.

Bottleneck Task Selection with a Multi-Resource Scheduler

We have extended CIRCA-II to consider multiple resources during plan scheduling. This work again focuses on augmenting the expressivity of scheduler-to-planner feedback to guide replanning when scheduling fails. However, we generalize on the "Schedule Manager" algorithms in two respects. First, we allow both multiple instances of a specific resource (e.g., multiple processors) as well as multiple classes of resources (e.g., processors, communication channels). Second, the above communication protocol between planner and scheduler were explicitly tied to the planning and scheduling algorithms used in CIRCA-II. We maintain a sufficiently expressive message-passing structure, but make the messages generic so that any planner and scheduler capable of generating and utilizing these messages can effectively be "plugged into" our planner-scheduler interface module.

We begin this section by introducing the real-time resource allocation and scheduling problem in terms of CIRCA-II. We then describe our heuristic planning-resource allocation interface, including how it may be "plugged into" CIRCA-II, and provide a simple example illustrating how this interface is used to select bottleneck tasks when scheduling fails.

Resource Allocation and Scheduling

For a real-time computing system, a "plan" is a set of tasks $T = \{T_1, \ldots, T_n\}$ with resource requirements and timing constraints. The problem of resource allocation is to map the set of planned tasks onto a set of available resources such that all constraints are met. In CIRCA-II, all guaranteed tasks are considered periodic, and each task $T_i \in T_{total}$ has worst-case computation time C_i ($wcet(T_i)$ previously) and period P_i. The worst-case computation time includes scheduler context-switching overhead. The *j*th invocation of task T_i becomes ready for execution at time $(j-1)P_i$, called task arrival time, $a_i[j]$. The deadline, $d_i[j]$, of a task invocation is usually such that $d_i[j] \leq a_i[j] + P_i$ since each invocation must complete its execution before the next one arrives. It is sufficient for the resource allocation algorithms to find a task schedule within a finite interval, *L*, equal to the least common multiple of all task periods, called the "planning cycle" in the real-time community. The resulting task schedule repeats itself in subsequent planning cycles. Each task may be composed of one or more separately schedulable modules (i.e., threads) with arbitrary precedence constraints. The resource requirements of each module are known *a priori* since we know the resource profile for the application code.

The selection of a proper resource allocation algorithm depends on the execution platform considered. An optimal resource allocation algorithm is described in [72] for uniprocessors, in [73] for multiprocessors, and in [56] for distributed systems. Once the task assignment is fixed, an optimal offline scheduling algorithm such as [3] can be used to preschedule the tasks. In this section, we presume the use of [56] for task assignment due to its ability to handle distributed systems, and use [3] as the "generic" periodic task scheduler. These algorithms are used to schedule all CIRCA-II guaranteed tasks (those with $g_i=1$). Best-effort tasks (those with $g_i=0$) are then fit, when possible, into gaps of this schedule. The resulting overall schedule for each processor is stored in a table.

Planning -- Resource Allocation Interface

A primary objective of the planning-resource allocation interface is to utilize existing resource allocation algorithms with minimal modification. In particular, the planner should be told whether or not the current plan is schedulable, and if it isn't, which task is judged to be the most costly "bottleneck". If the plan is found schedulable by the resource allocation analyzer then its entire value is redeemed. However, if the plan is unschedulable, the interface module points out a "costly" task to reconsider during replanning.

In our design, we require a specific plan format for transmission to the planner-scheduler interface module. As described previously, the CIRCA-II planner produces a set of TAPs, a subset of which are guaranteed to preempt *ttfs* along with the rest which operate under strictly best-effort operation. Henceforth in this section we shall refer to each TAP as a task T_i. For each planned task $T_i \in T_{total}$, where T_{total} contains all tasks in the plan, the planner must output the triplet (g_i, P_i, V_i) to the planner-scheduler interface module. g_i is the "guarantee flag" that indicates whether task T_i is guaranteed $(g_i = 1)$ or best-effort/"if-time" $(g_i = 0)$. P_i is the maximum period[49] of T_i required to preempt *ttfs* when $g_i = 1$, and V_i is the "priority" value of task T_i, currently set to $n_i * max(prob_i)$, where n_i is the number of reachable states in which task T_i executes and $max(prob_i)$ is the maximum probability of any state in which T_i executes. This heuristic reflects a preference to keep tasks chosen for the highest-probability states, as well as the fact that large n_i will likely require many backtracking steps should T_i be altered. We define the set $T_{mandatory}$ as all tasks $T_i \in T_{total}$ with $g_i = 1$.

[49] For the CIRCA-II planner, P_i must be set to twice the TAP separation constraint (t_{sep}) because periodic schedulers may place a task either at the "beginning" or "end" of each task period within a schedule, as described in [52]. This will often result in inefficient scheduling of CIRCA-II plans, but is more generic due to the multitude of periodic task allocation and scheduling algorithms.

The resource allocation analyzer receives input *($T_i \in T_{mandatory}$, P_i)* then returns a success/failure status and utilization matrix U in which each element *$U(i,q)$* is the utilization consumed by task T_i of resource class q. The scheduler database defines the worst-case resource requirements of all task modules (or threads) $M_j \in T_i$. Elements *$U(i,q)$* are computed by the resource allocation analyzer as follows. Within each planning cycle L the total capacity of a resource q is pQL, where p is the number of instances of the resource and Q is the capacity of each. If module M_k of period P_k and execution time C_k requires an amount $r_{k,q}$ of the resource throughout its execution, then its total demand on that resource within the planning cycle is $r_{k,q}C_kL/P_k$. The ratio of that demand to the total available resource capacity is the utilization *$u(k,q)$* consumed by module (or thread) M_k of resource q, and is shown below in Equation 6-2. The utilization *$U(i,q)$* consumed by task T_i of resource q is the sum of the utilizations *$u(k,q)$* of all modules $M_k \in T_i$ and is shown below in Equation 6-3.

$$u(k,q) = \frac{r_{k,q}C_k}{pqP_k} \tag{6-2}$$

$$U(i,q) = \underset{\forall k, M_k \lfloor T_i}{} u(k,q) \tag{6-3}$$

To compute the most "costly" task in cases of over-utilization (failure), the interface combines priorities V_i from the planner with the utilization matrix U from the resource allocation analyzer. The interface module tentatively deletes one action, T_j, from the plan and recomputes the resulting aggregate utilization *$\gamma_j(q)$* by adding *$U(i,q)$* for all *$i \neq j$*. The bottleneck resource *$q_b(j)$* for task T_j is the one for which *$\gamma_j(q)$* is maximum, as shown in Equation 6-4.

$$q_b(j) = max_q(\gamma_j(q)) = max_q(\underset{i,i?j}{} U(i,q)) \tag{6-4}$$

The total value Sum_j remaining after eliminating T_j is the sum of V_i, as shown in Equation 6-5. The total value per unit of bottleneck-resource-usage is thus $Sum_j / \gamma_j(q_b(j))$. The interface recommends removal of the action $T_{bottleneck}$ that results in the maximum value per cost ratio, as shown in Equation 6-6. Note that the Sum_j defined here is not exact, since removal of one action could affect the V_i values of other actions. Exact computation of Sum_j would require detailed knowledge of the planning state-space after this action was removed, which is time-consuming.

$$Sum_j = \underset{i,i?\,j}{} V_i \qquad (6\text{-}5)$$

$$T_{bottleneck} = max_j \left(\frac{Sum_j}{\gamma_j(q_b(j))} \right) \qquad (6\text{-}6)$$

This heuristic is used to suggest which part of the planner's search space to expand next, via dynamic backtracking to each state in which $T_{bottleneck}$ was guaranteed to preempt a *ttf*. However, it does not actually prune parts of the search space. Since the planner's search is exhaustive in the worst-case, it will find a feasible plan if one exists. Much like with the "Schedule Manager" utilization feedback, this "bottleneck task selection" heuristic merely increases the odds of finding such a plan earlier in the search process.

Example: Selecting a Bottleneck Task in an Unschedulable Plan

We illustrate the computation of $t_{bottleneck}$ with a simple example in this section. Assume the plan as downloaded from the planner consists of four tasks, $T_{total} = \{T_1, T_2, T_3, T_4\}$, and that all of these tasks are guaranteed $(g_i=1)$ since best-effort tasks are not considered by the interface module. Further, assume all tasks have priority value $V_i=1.0$ for simplicity, thus the bottleneck task will be determined strictly from utilization considerations. Let Table 6-2 describe the utilization matrix *U(i,q)* returned from the resource allocation/scheduling module, where the columns represent utilization values for the four guaranteed tasks and the rows represent utilization values for the three available resource classes. As can be observed from this matrix, at least resource q_3 is overutilized since the sum of individual task utilizations is greater than 1. Thus, scheduling has failed and a bottleneck task must be identified.

Table 6-2. Example Utilization Matrix *U(i,q)*.

	T_1	T_2	T_3	T_4
q_1	0.1	0.35	0.4	0.05
q_2	0.25	0.3	0.1	0.15
q_3	**0.15**	**0.4**	**0.25**	**0.3**

Table 6-3. Values used for Computation of $T_{bottleneck}$.

	T_1 removed	T_2 removed	T_3 removed	T_4 removed
$\gamma_j(q_1)$	0.8	0.55	0.5	0.85
$\gamma_j(q_2)$	0.55	0.5	0.7	0.65
$\gamma_j(q_3)$	0.95	0.7	0.85	0.8
$q_b(j)$	q_3	q_3	q_3	q_1
value/cost	3.16	***4.29***	3.53	3.75

Table 6-3 shows the aggregate utilization values $\gamma_j(q)$ and bottleneck resource $q_b(j)$ after the removal of each task T_j as computed from Equation 6-4. In this example, the value Sum_j remaining after eliminating any one task is always the same (equal to 3, the number of tasks remaining, since $V_i=1$ for all tasks). Table 6-3 also shows the value-to-cost ratios after removal of each task. The maximum value-to-cost ratio remains after removing task T_2, thus $T_{bottleneck} = T_2$, which is then fed back to the planner for dynamic backtracking.

The reader may question why we include the previous "Schedule Manager" section in this thesis, given that this new planner-scheduler interface module functionally subsumes the single-processor Schedule Manager. One compelling reason is that the Schedule Manager is fully-implemented while only the Planning Subsystem CIRCA-II component for this model is sufficiently implemented to test the feasibility of our planner-scheduler interface heuristic and message-passing algorithms. To experimentally verify the multi-resource scheduler-to-planner feedback proposed here, CIRCA-II's Plan Execution Subsystem would require substantial redesign to accommodate multiple resources and handle resource failures (see below). The simple part of multi-resource plan execution is running a single plan on multiple resources as prescribed by the multi-resource scheduler.[50] However, it will be non-trivial to implement the plan dispatcher such that it can process feedback from any execution resource then efficiently switch plans uniformly across all resources.[51] Then, to continue experimental verification of the fault-tolerance procedure outlined below, we will also have to implement resource monitoring procedures as "feature sensing" actions, as well as the capability to execute tasks on any subset of operational resource instances. We are currently working to address these implementation challenges within CIRCA-II's Plan Execution Subsystem and hope to have a preliminary multi-resource, fault-tolerant system design and prototype that operates on the QNX real-time operating system within the next few months.

[50] We presume homogeneous resource instances, so each task can access feature values and execute actions from any particular instance of a resource.

[51] In fact, the plan dispatcher should itself be distributed across multiple processors for maximally-efficient resource utilization. Otherwise, if a processor containing the dispatcher fails, no further plan switches will be possible thus the system will not even be capable of retrieving the plan to handle the "processor failure" fault, as described next.

"Internal" Fault-Tolerance during Plan Execution

Both the original CIRCA and CIRCA-II are designed to accommodate any "external faults" in the world that can be modeled with as sensed features and handled via some action on the environment. However, neither system can consider the possibility of "internal" computational resource faults so long as processes require a fixed set of resources as was the case with a single-processor scheduler and plan-execution system. With the introduction of the multi-resource allocation and scheduling procedures described above, we can also begin to introduce the notion of tolerance to computational resource failures in CIRCA-II, specifically to the critical plan-execution platform which must execute plans reliably and in hard real-time to avoid any possible catastrophic system failures. In this section, we first describe how we build upon the planner-scheduler interface to develop plans that exhibit tolerance to a limited user-specified set of faults, and then give an automated aircraft example that illustrates the utility of specifically designing plans to accommodate computational resource failures.

Developing Fault-Tolerant Plan Sets

We establish tolerance to "internal" computational resource faults (e.g., single processor failures) by using the planner-resource allocation interface module to effectively manage a preset list of faults for which the system must be tolerant. This list, F_{total}, is specified by the user as part of the scheduler knowledge base. It includes the nominal "no-fault" case f_0 in which all computational resources work properly, and progressively describes more severe faults, terminating with the worst fault f_n the system must tolerate.

The CIRCA-II multi-resource allocation and scheduling system access a description of the available resource types and quantities for each fault $f_i \in F_{total}$ from the scheduler database. In this manner, the allocation and scheduling processes will schedule each plan in accordance with the actual set of fully-functional resources that would be available given a specific fault condition instead of presuming that the complete, fixed set of computational resources are always available for plan execution.

Figure 6-3 shows the planner-scheduler interface algorithm used to control all data flow between the CIRCA-II planner and scheduler, including both "nominal" plan development as described in the previous section and all additional steps required to develop the set of fault-tolerant plans required to handle each fault $f_i \in F_{total}$. To summarize, the interface module incorporates plan and utilization data for each fault to classify plans as "good" or "unschedulable". A good plan is added to F_{good}, then downloaded to the plan dispatcher along with indices to all faults for which that plan was "good". These faults are removed from the working fault list (i.e., placed in F_{done}), since they only require one plan. For the first (i.e., least severe) fault that over-utilized resources, a "bottleneck" task is recommended for removal using the heuristic described in the previous section, then fed back to the planner which backtracks to find a safe alternative plan. This procedure continues until all faults have been handled successfully by some schedulable plan, even if safety guarantees for the final most-severe faults in F_{total} are only probabilistic (with perhaps even a decent chance of failure).

This algorithm enables creation and storage of (i) a set of plans that can meet all required hard real-time constraints when any internal fault from F_{total} occurs, and (ii) a pre-computed execution schedule for each plan. After the plan dispatcher fills the cache with "good" plans for all faults, the plan indexed for the nominal no-fault condition f0 (for the first planned subgoal) is selected and begins execution according to the computed schedule. When the system detects an internal fault,[52] plan execution switches to the pre-

[52] As was noted earlier, we are still in the process of implementing the CIRCA-II multi-resource plan-execution platform, thus we also still have to build fault-detection monitors into our software.

scheduled plan designated to handle that fault. Thus, response to internal faults is prompt, and the system does not fail due to internal faults except when a computational fault occurs that is outside the limited set F_{total}.

```
1. Plan $T_{total}$ is received; each $T_i \in T_{total}$ is specified by the triplet $(g_i, P_i, V_i)$.
2. $\forall (f_i \in F_{total}; f_i \notin F_{done})$
    -- Send $(P_j$ for all $T_j \in T_{mandatory}, f_i)$ to scheduler, which returns matrix $U(j,q)$.
    -- If scheduling succeeds, add $f_i$ to list $F_{good}$ for plan $T_{total}$; add $f_i$ to $F_{done}$.
3. If $(F_{good} \neq \emptyset)$, download $T_{total}$ with indices $F_{good}$ to dispatcher; reset $F_{good} = \emptyset$.
4. If $(F_{done} \neq F_{total})$,
    -- Find first element $f_i \in F_{total}$ such that $f_i \notin F_{done}$.
    -- Send to planner "bottleneck" task $T_{bottleneck}$ identified from Equation 6-6.
    -- Go to Step 1.
```

Figure 6-3: Planning-Scheduling Interface with Fault-Tolerant Plan Development.

Autonomous Aircraft Example

We consider an example from automated flight to illustrate the utility of plan development with the fault list. Our plan execution system in this example includes two resource types: *Proc* (processor) and *Comm* (communication channel). The system contains two processors of type *Proc* and a single communication channel of type *Comm*, and we define a fault set which includes the nominal no-fault case (f_0) and a "single processor failure" fault (f_1), in which the number of *Proc* instances is reduced from 2 to 1.

For our automated flight mission, the CIRCA-II planner is given the goals of maintaining safety while following a specific flight trajectory. The aircraft must follow standard air traffic procedures and maintain communication with Air Traffic Control (ATC) via the *Comm* channel resource, which we assume to have guaranteed worst-case execution properties. For this example, we present a very simplified aircraft world model which illustrates how safety is maintained during flight, even in the presence of a single processor failure from the set of *Proc* resources.

In its initial phase, the planner builds the state set shown in Figure 6-4. In this plan, two failures must be avoided: an *impact* with an obstacle (e.g., the terrain or another aircraft) and any *airspace-violation* (e.g., flying in a restricted military area). To prevent these *ttfs*, CIRCA-II selects two actions: *avoid-collision* and *maintain-trajectory*.

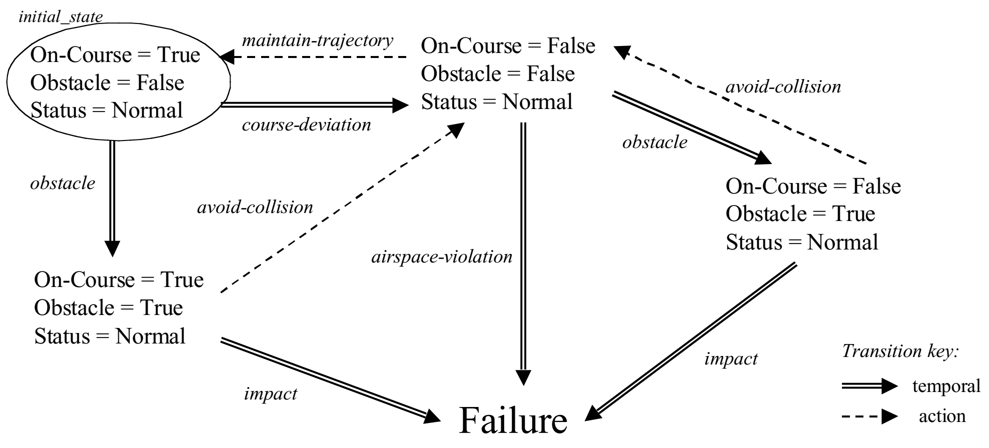


Figure 6-4: Nominal Flight Plan.

Table 6-4: Flight Example Task Set.

T_i	P_i	V_i	*Modules*
avoid-collision *(T_1)*	6 (nominal plan)	1	M_1, M_2, M_3
maintain-trajectory *(T_2)*	12 (nominal plan)	1	M_4, M_5
declare-emergency *(T_3)*	6 (reduced plan)	1	M_6
follow-radar-vectors *(T_4)*	12 (reduced plan)	1	M_7, M_5

The decomposition of all tasks available in our flight example is shown in Tables 6-4 and 6-5. To detect a state with *Obstacle=True*, task T_1 runs modules M_1, *scan-TCAS* (Terminal Collision and Avoidance System), to sense nearby obstacles and M_2, *monitor-traffic*, to detect other air traffic based on ATC data. If an object is detected, the avoid-obstacle action is executed. The *maintain-trajectory* task *(T_2)* executes to detect course deviations with M_4, *monitor-course*, and correct them by sending reference trajectory *(r(t))* commands to the low-level controller via M_5, *update-controller-reference*.[53]

Table 6-5: Flight Example Module Worst-Case Resource Usage.

Module	*Function*	C_i on *Proc*	C_i on *Comm*
M_1	scan-TCAS	2	-
M_2	monitor-traffic	3	2
M_3	avoid-obstacle	4	-
M_4	monitor-course	4	-
M_5	update-reference	4	-
M_6	declare-emergency	1	1
M_7	receive-vectors	2	5

Table 6-4 also includes the period *(P_i)* and priority *(V_i)* used by the scheduling and planner-scheduler interface algorithms. For this example, we again set all task priorities equal *($V_i=1$)*. Note that all actions are guaranteed *($g_i=1$)* since all states with planned actions have *ttfs*. The computing system is composed of two processors, each a

[53] As will be discussed in Chapter 8, CIRCA-II relies on a traditional low-level control system to read sensors and compute actuator commands. This controller is presumed to have its own set of fault-tolerant resources since it is always required for autonomous operation.

resource of type *Proc*, interconnected with each other and ATC by a communication bus, a resource of type *Comm*. Once CIRCA-II has developed the initial plan, the allocation/scheduling system attempts to schedule it for each specified fault $f_i \in \{f_0, f_1\}$. For the no-fault case *(f_0),* a valid task assignment [56] and schedule [3] is computed such that all constraints are met. The resource allocation for f_0 is shown in Figure 6-5. This schedulable plan *(Plan1)* for the nominal no-fault case f_0 is now added to the "good" list, F_{good}, and f_0 is added to the set of handled failure modes F_{done}.

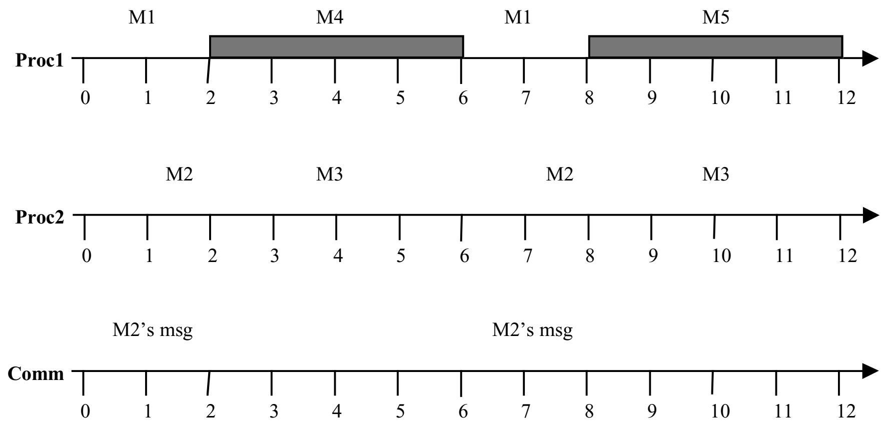


Figure 6-5: Nominal Plan Resource Schedule *(f_0).*

As shown in Table 6-6, the processor (Proc) utilization exceeds a value of one for f_1, thus the initial plan must be altered for f_1. The planner-scheduler interface uses utilization matrix feedback to recommend that the planner remove T_1 (avoid-collision) due to its high *Proc* utilization. Backtracking during replanning yields the state diagram shown in Figure 6-6, with the new task declare-emergency (T_3) selected.[54] Once the emergency is declared, ATC effectively takes much of the computational responsibility from the aircraft, clearing airspace so that obstacles will no longer be a factor.

[54] All states from the nominal plan *(Plan1)* are possible. The temporal transitions *obstacle* and *course-deviation* are not preempted since they may happen too quickly.

Additionally, after an emergency has been declared, the efficient action follow-radar-vectors can be selected, in which ATC specifies the course and corrections required for the aircraft to safely reach its destination.

Table 6-6: Utilization Matrix for the Nominal Plan.

T_i	$U(i, Proc, f_0)$	$U(i, Proc, f_1)$	$U(i, Comm)$
T_1	14/24	14/12	4/12
T_2	8/24	8/12	0/12

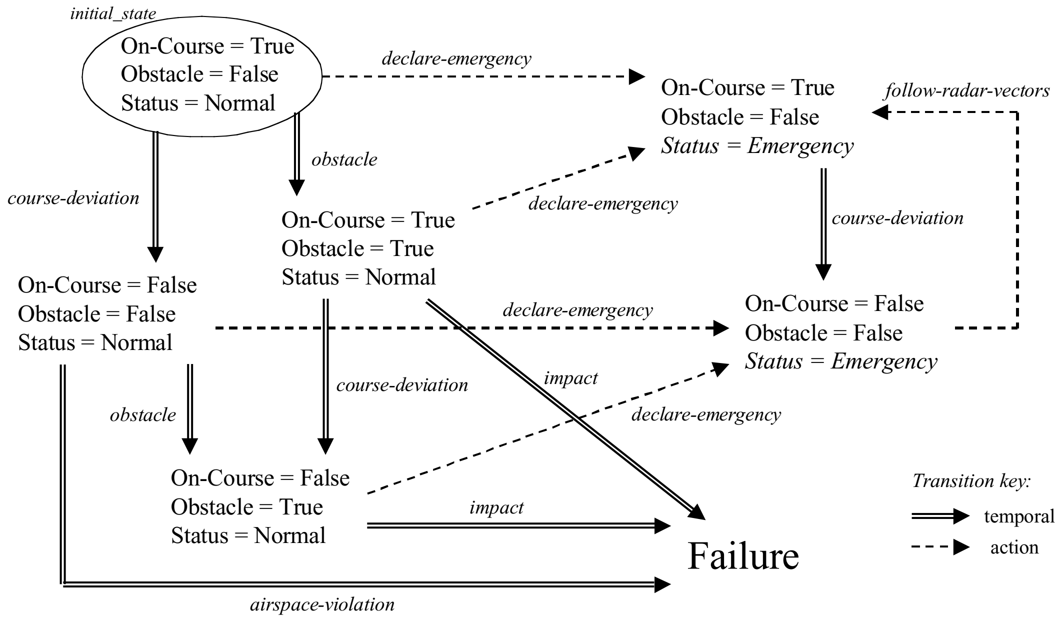


Figure 6-6: Reduced Flight Plan for Failed Processor *(f1).*

This reduced plan (*Plan2*) is now sent to the resource allocation/scheduling module, which finds the plan can easily be scheduled even with the processor failure (f_1), as computed with task utilizations shown in Table 6-7 and a valid task assignment illustrated in Figure 6-7. With this plan, CIRCA-II can handle both f_0 and f_1, so *Plan2* is stored and planning (for this subgoal) terminates.

Table 6-7: Utilization Matrix for the Reduced Plan.

T_i	$U(i, Proc, f_1)$	$U(i, Comm)$
T_3	2/12	2/12
T_4	6/12	5/12

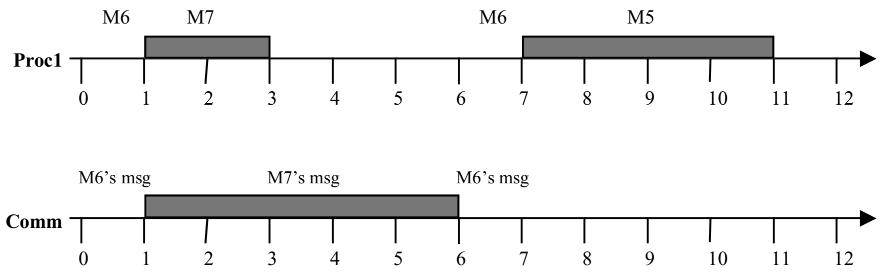


Figure 6-7: Reduced Plan Resource Schedule *(f1).*

In this example, we have identified unschedulable plans and made them schedulable via replanning. This is in contrast to traditional resource allocation algorithms which simply fail if a plan is unschedulable. It also contrasts with planning algorithms which do not consider failures of computing resources, and do not guarantee schedulability of plans in a hard real-time sense.

QoS Negotiation during Scheduling[55]

In CIRCA-II, we have presumed that each task module has fixed execution properties, thus each has inflexible worst-case resource requirements that must always be used by the scheduler. The real-time community has recognized that many functions may

[55] This work was done cooperatively with T. F. Abdelzaher. Further details of the QoS Negotiation protocol, including its incorporation into middleware services called RTPOOL, are provided in [1] and [2].

be written with the flexibility to trade off result accuracy with resource requirements. Furthermore, often the relative accuracy may be captured numerically so that software can subsequently trade off the efficacy of degrading one software module versus another. The *Quality-of-Service (QoS) negotiation* research field has been introduced to develop principled ways of degrading software to reduce its resource requirements for real-time systems. We have begun work to study how QoS-level adaptation to can be used to relax scheduling constraints on hard real-time tasks with a focus on tailoring QoS negotiation algorithms to fit within future implementations of CIRCA-II.

As discussed previously, whenever a proposed plan is found to be unschedulable, CIRCA-II has to-date placed the burden of creating a schedulable plan solely on the planner. We hope to incorporate QoS adaptation techniques into CIRCA-II specifically so that the scheduler can use a table of QoS levels for each thread/module to trade off task execution requirements (e.g., worst-case execution times) with execution performance (e.g., result accuracy). Then, if scheduling fails when assuming all highest (best-performing) QoS levels as is currently done in CIRCA-II, the scheduler can degrade these QoS levels and propose alternate schedules that execute sufficiently but not optimally. This process will have two benefits. First, feedback from scheduler to planner can be made even more expressive by letting the planner know exactly how much performance must degrade to make the plan schedulable, based on the relative cost associated with decreasing each QoS level. This is in contrast to totally removing a "conflicting" hard real-time task (with the Schedule Manager) or selecting an ordering for bottleneck tasks (with the planner-scheduler interface module) to be used during planner backtracking. Additionally, should the planner be completely unable to build a schedulable plan when all tasks assume their absolute worst-case execution times, reducing task module QoS levels may enable plan scheduling with only a minimal degradation in performance, whereas previously the system would either fail or assume an unrealistically large probability threshold below which all states were ignored.

In this section, we first describe QoS negotiation and the heuristic we use to decide how QoS levels will be degraded during scheduling, then present an example illustrating how graceful performance degradation may be achieved when using this algorithm for autonomous flight control.

QoS Negotiation Protocol

For this discussion, we will generalize beyond CIRCA-II to consider a group of clients making requests on a platform which must schedule these requests on its available resources. We map CIRCA-II into this framework by considering a "client" to be a task *(T_i)* that requires a group of threads to be executed with a specified maximum period *(P_i)*. Thus, a complete CIRCA-II plan specifies a set of "clients" that simultaneously send resource requests to a scheduler.

Our QoS negotiation model is centered around three simple abstractions: *QoS levels, rewards,* and *rejection penalty*. A client requesting service specifies in its request a set of negotiation options and the penalty of rejecting the request derived from the expected utility of the requested service (i.e., task execution). Each negotiation option consists of an acceptable QoS level for the client to receive from the provider and a reward value commensurate with this QoS level. The QoS levels are expressed in terms of parameters whose semantics need to be known only to the client and service provider. For example, in processor capacity reservation, they may express the required processor bandwidth, while in a multicast protocol they may represent the semantics of the requested multicast service, such as reliable, ordered, causal, or atomic delivery. The *reward* represents the "degree of satisfaction" to be achieved from the QoS level (i.e., application-perceived utility of supplying the client with that level of service).[56] Thus,

[56] When applying these QoS Negotiation techniques to CIRCA-II, *reward value* for each thread QoS level will probably be based on a fraction of overall performance degradation

the client's negotiation options represent a set of alternatives for "acceptable" QoS and their "utility". The *rejection penalty* of a client's request is the penalty incurred to the application if the request is rejected.[57]

To control system load in a way that ensures predictable service, a scheduler must determine whether the request can be guaranteed or must be rejected. We propose a slightly different notion of guaranteeing a request, as compared to the conventional notion of a guarantee. In our proposed model, guaranteeing a client's request is the certification of the request to receive service at one of the QoS levels listed in its negotiation options. The selection of the QoS level it will actually receive, however, is up to the scheduler. Furthermore, during plan execution, the service provider will be free to switch this QoS level to another level in the client's negotiation options if it increases perceived utility (e.g., if tasks take much less than their worst-case requirements, the dynamic scheduler can insert one or modules with a higher QoS level to take advantage of the extra slack resources).

The real-time group at the University of Michigan has designed a middleware service, RTPOOL [1], that has been used to illustrate the utility of our QoS Negotiation protocol.[58] Each incoming client request for scheduling a task includes its rejection penalty along with the different QoS levels and their rewards. A client task's QoS level is specified by the parameters of its execution model. For an independent periodic task, the parameters consist of task period, deadline, and execution time. We model period and

---

relative to the highest QoS level for that module scaled by the "priority" for the task containing this module. However, we have not yet formalized this computation.

[57] *Rejection penalty* may be set to *$(1/V_i)$* for CIRCA-II best-effort tasks.

[58] RTPOOL is more general than the CIRCA-II scheduler in that it is capable of dynamically receiving requests from a variety of different clients while other requests are already executing, but since RTPOOL effectively subsumes the functionality of the CIRCA-II scheduler, its algorithms and their functionality will be transferable to CIRCA-II in future implementations.

deadline as negotiable parameters for all soft real-time tasks. Task execution time, on the other hand, depends on the underlying machine speed and thus should not be hardcoded into the client's request. Instead, each QoS level in the negotiation options specifies which modules of the client task are to be executed at that level. This allows the programmer to define different versions of the task to be executed at different QoS levels, or to compose tasks with mandatory and optional modules. The reward associated with each QoS level tells RTPOOL the utility of executing the specified modules of the task with the given period and deadline.

When one or more new requests arrives at a machine, RTPOOL executes a local QoS optimization heuristic, which computes the set of QoS levels for all local clients to maximize the sum of their rewards. Tasks are inserted into the schedule upon arrival order (or randomly if multiple arrive simultaneously). A task may be rejected if both (i) the new sum of rewards (including that of the newly-arrived task) is less than the existing sum prior to its arrival, and (ii) the difference between the current and previous sums is larger than the new task's rejection penalty. Otherwise, the requested task is guaranteed. As a result, task execution requests will be guaranteed unless the penalty from resulting QoS degradation of other local clients is larger than that from rejecting the request. When a task execution request if rejected by the local machine, one may attempt to transfer and guarantee it on a different machine using a load-sharing algorithm. Note that conventional admission control schemes would always incur the request rejection penalty whenever an arrived task makes the set of current tasks unschedulable. By offering QoS degradation as an alternative to rejection and by using admission control rules, we can show that the reward sum (or perceived utility) achieved with our scheme is *lower bounded* by that achieved using conventional admission control schemes given the same schedulability analysis and load sharing algorithms. Thus, in general, our proposed scheme achieves higher perceived utility.

Figure 6-8 gives an example of a local QoS optimization heuristic. The heuristic implements a gradient descent algorithm, terminating when it finds a set of QoS levels that keeps all tasks schedulable, if any. Note that unless all tasks are executed at their highest QoS level, the machine suffers from *unfulfilled potential reward*. The unfulfilled potential reward, UPR_j, on machine N_j, is the difference between the total reward achieved by the current QoS levels selected and the maximum possible reward that would be achieved if all local tasks were executed at their highest QoS level. This difference can be thought of as a fractional loss to the mission and is often unavoidable due to resource limitations. However, such loss may also be caused by poor load distribution, in which case it can be improved by proper load sharing.

```
Let each client task $T_i$ have QoS levels $M_i[0],…,M_i[best_i]$ with rewards
$R_i[0],…,R_i[best_i]$, respectively.
1. Start by selecting the best QoS level, $M_i[best_i]$, for each client $T_i$.
2. While the set of selected QoS levels is not schedulable, do Steps 3
   and 4.
3. For each client $T_i$ receiving service at level $M_i[j] > M_i[0]$, determine
   the decrease of local reward, $R_i[j]-R_i[j-1]$, resulting from degrading
   this client to the next lower level.
4. Find client $T_k$ whose $R_k[j]-R_k[j-1]$ is minimum and degrade it to the
   next lower level.
5. Go to Step 2.
```

Figure 6-8: Local QoS Optimization Heuristic.

RTPOOL employs a load-sharing algorithm that implements a distributed QoS-optimization protocol. Described in Figure 6-9, the protocol uses a hill-climbing approach to maximize the global sum of rewards across all clients in the distributed pool. It is activated between two machines N_i and N_j when the difference UPR_i - UPR_j exceeds a threshold V_{thresh}. Close examination of the local QoS optimization heuristic and the distributed QoS optimization protocol reveals that neither makes assumptions about the nature of the client and the semantics of its QoS levels. The distributed QoS-negotiation protocol, however, assumes service to a given client can be migrated to another node. In the near future, we hope to incorporate QoS negotiation algorithms from RTPOOL into the CIRCA-II multi-resource scheduling system described previously. Below, we illustrate how a scheduler may automatically degrade task QoS levels for a flight control system. In this example, we do not alter the actual tasks, but rather their execution properties, thus the CIRCA-II planner need not re-plan to find a solution.

```
1. On source machine $N_i$ find client $T_k$ whose removal will result in max.
   increase, $W$, in total reward.
2. $N_i$ requests reassigning $T_k$ with reward $W$.
3. Each machine $N_j$, where $UPR_i - UPR_j > V_{thresh}$, receives the request and
   recomputes QoS levels for its local clients plus $T_k$.  If its total
   reward is higher with $T_k$, $N_j$ bids for $T_k$ with the reward increment $W_j$
   resulting from accepting it.
4. $N_i$ transfers $T_k$ to the highest bidder.
```

Figure 6-9: Distributed QoS Optimization Protocol.

Example of QoS Negotiation for Autonomous Flight

We have performed a preliminary test of our QoS negotiation protocol with an aircraft simulation, which will be described in more detail in Chapter VII. Our QoS-negotiation scheme enables the application domain expert (or the CIRCA-II planner via its knowledge base) to express application-level semantics using QoS levels, rewards, and rejection penalty. Table 6-8 shows the minimal set of tasks we used to control the aircraft during a short flight in which we were to destroy any observed enemy targets using the simulated F-16's onboard radar and missiles. Four separate tasks were required to control the aircraft: Guidance *(Guid),* Control *(Ctrl),* "Slow" Navigation *(Snav),* and "Fast" Navigation *(Fnav).* These tasks function much like their similarly-named Flight Management System counterparts. *Guid* is responsible for computing the reference trajectory state for the aircraft. In our tests, *Guid* specified only heading and altitude to lead the aircraft along its trajectory. The *Ctrl* task is responsible for executing the low-level control loops to compute actuator commands from the high-level guidance trajectory. We have two navigation tasks *(Snav, Fnav)* to estimate aircraft state, distinguished by required update frequency. More details on our specific control laws may be found in Chapter VII.

Table 6-8: QoS Levels for the Automated Flight Control Plan.

Task	*L*	*R*	*ET(ms)*	*P(sec)*	*Ver*
Guid	0	10	100	10	*def**
	1	15	100	5	*def*
	2	20	100	1	*def*
Ctrl	0	1	80	5	*sec***
	1	100	60	1	*prim*
	2	104	80	1	*sec*
	3	120	60	0.2	*prim*
	4	124	80	0.2	*sec*
Snav	0	10	100	10	*def*
	1	20	100	5	*def*
	2	25	100	1	*def*
Fnav	0	1	60	5	*def*
	1	100	60	1	*def*
	2	120	60	0.2	*def*
MC	0	1	500	10	*def*
	1	30/200	500	1	*def*

**def* is the default version to execute for each task.
**For the *Ctrl* task, two versions are available, one that uses only primary actuators *(prim)* and another *(sec)* that allows higher-performance control through the use of secondary actuators (e.g., afterburners).

Table 6-8 also shows the QoS levels *(L)* present for all tasks, including associated rewards *(R),* execution time *(ET),* period *(P),* and version *(Ver).* In our simple tests, we set each task deadline equal to its period, as would be required by the CIRCA-II planner, although there would be no such requirements for our generic QoS negotiation protocol. Also, because all tasks are considered critical to execute (at least at a degraded QoS level), we set all task rejection penalties sufficiently high that all tasks are always accepted by the QoS negotiator.

In addition to the basic flight control tasks, we simulate a function necessary during military operations: Missile Control *(MC)*. *MC* is composed of two precedence-constrained threads: *read-radar* and *fire-missile*. *Read-radar* monitors aircraft radar to detect approaching enemy targets, then *fire-missile* launches a missile at any enemy targets appearing on radar. As shown in Table 2, *MC* is computationally expensive and has two QoS levels. In Level 1, radar will be scanned with sufficient frequency to allow detection and destruction of most enemy targets. Otherwise (Level 0), fast-moving targets may not be destroyed. During experiments, we varied the reward for *MC* QoS Level 1 depending on the relative importance of destroying enemy targets.

QoS Negotiation Example Evaluation

In this section, we show results that illustrate how QoS negotiation can help aircraft flight control degrade gracefully. First, we assess the QoS negotiation heuristic for our flight asks by observing how the QoS of each task degrades with lower machine speeds. Next, we look at aircraft performance during flight as a function of the *Ctrl* task's QoS level, and conclude with tests using the *missile-control* task to observe the effects of load sharing between two machines, with processor failure used to demonstrate graceful performance degradation.

Our gradient-descent-based local QoS optimization heuristic was designed to help a service provider select a high-reward set of QoS levels for its clients. Using the QoS levels and rewards listed in Table 6-8, we illustrate the behavior of the presented heuristic. In this experiment we kept the task set fixed, and decreased the underlying CPU speed (increasing task execution times), then observed the corresponding decrease in task QoS levels. Figure 6-10 plots QoS levels (modes) selected vs. CPU speed, normalized by the minimum CPU speed for which the task set is schedulable. Since the heuristic uses only reward information to guide its search for a feasible QoS level set (thus being applicable as-is in any service that uses our QoS negotiation scheme), optimality is compromised yet "graceful QoS degradation" is still illustrated.

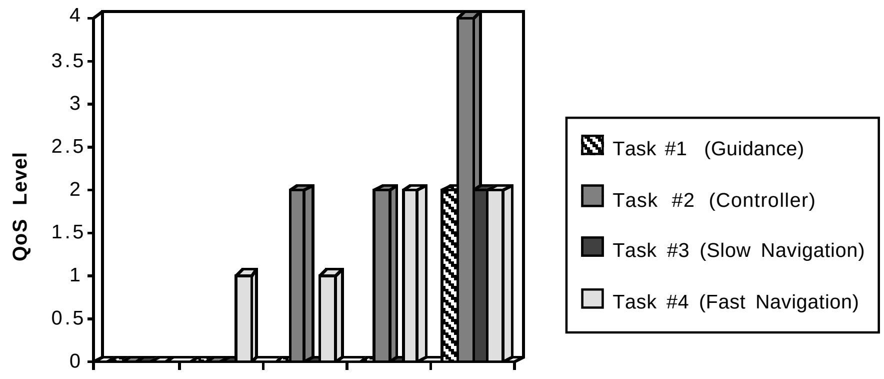


Figure 6-10: QoS Levels vs. CPU Speed for Flight Control Tasks.

For our next set of experiments, we evaluated system performance by studying its ability to control the simulated F-16 during flight. All flight control tasks were executed on one processor. As shown in Table 6-8, Ctrl QoS levels are a function of both task period and version. We present tests that show flight performance differences due to each of these variables, specifically during the critical takeoff/climb phase of flight. Figure 6-11 illustrates differences between the two versions of *Ctrl* in their "best performance" case (P_i = 200 msec). Level 4 (with secondary actuation) requires larger execution time than level 3 (primary actuation only), thus is harder to schedule. Climb performance for level 4 is only slightly better for level 3, consistent with their small reward difference. This example illustrates how QoS negotiation can achieve graceful degradation. Overall processor utilization is decreased by reducing *Ctrl* to level 3, but safety (controller stability) is not compromised.

Next, we performed tests with varying *Ctrl* task period. We isolated version from period effects by exclusively selecting QoS levels with secondary actuation, although similar trends result with the other *Ctrl* version (levels 1 and 3). To illustrate performance changes as a function of task period, we consider three QoS levels: level 4 (P_i = 0.2 sec), level 2 (P_i = 1 sec), and level 0 (P_i = 5 sec). We include level 0 among *Ctrl* QoS negotiation options as a comparative example illustrating controller instability. Of course, no unstable QoS levels should be defined among a client's negotiation options, since the client should not "ask" for instability.

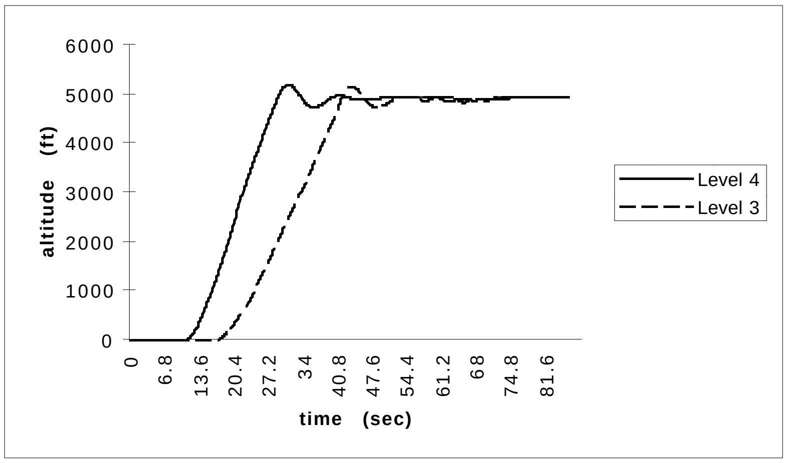


Figure 6-11: Altitude with and without Secondary *Ctrl* Actuation.

Figures 6-12 through 6-15 show aircraft state as a function of time during takeoff, climb, and a 90° turn. Figure 6-12 shows aircraft altitude for the different Ctrl task periods. As period increases, climb performance gracefully degrades between QoS levels 4 and 2, but then becomes unstable in level 0 (P_i = 5 sec), illustrating the necessity of real-time response for the Ctrl task. Figures 6-14 and 6-15 show aircraft pitch and roll angle, respectively, for the "stable" controller QoS levels. Note that we do not include level 0 because the unstable response obscures the other plots. Pitch angle and altitude are coupled, so pitch has largest magnitude during the climb, and as illustrated, the period increase to 1 second causes a large pitch angle to be required longer, a stable (for this gentle-maneuver flight) but undesirable trait. Roll angle shows delay and longer deviation from zero as well as significant overshoot when task period increases.

Load sharing capabilities were studied in a final test set which included both the flight control tasks and the missile-control task. We started the system with two processors available for task execution, thus the scheduler populates them accordingly. In this configuration, the load sharing protocol places all flight control tasks on one machine and the missile-control task on the other processor.

When the two machines function normally, both flight and missile control (*MC*) asks run at their maximum QoS levels. In this case, enemy targets are quickly detected and fired upon, while flight control is identical to the best performance profiles in Figures 6-12 through 6-15. However, when a processor failure occurs (analogous to the occurrence of fault *f1* in the earlier "multi-processor planner-scheduler interface" example), the load sharing and QoS negotiation protocols (as implemented in RTPOOL) dynamically adjust task QoS levels such that all tasks can fit on one machine. If *MC* is assigned relatively low reward (the value 30 from Table 6-8), the system degrades *MC*, *Guid*, and *Snav* tasks but keeps *Ctrl* and *Fnav* tasks at high levels. In this manner, flight control is a bit sluggish but stable, but the aircraft is unable to launch missiles at most targets. Alternatively, this system may be aboard an expendable drone whose most important function is to destroy a target or attack enemy aircraft. In this case, the reward set may be structured such that *MC* takes precedence over accurately maintaining flight control. For a drone, we assign relatively high reward to *MC* (the value 200 from Table 6-8), and when a processor fails, the QoS negotiator reduces all flight control tasks to QoS level 0 while maintaining the level of the "important" *MC* task. Thus, the aircraft eventually becomes unstable and crashes, but will quickly detect and respond to enemy targets on its kamikaze mission.

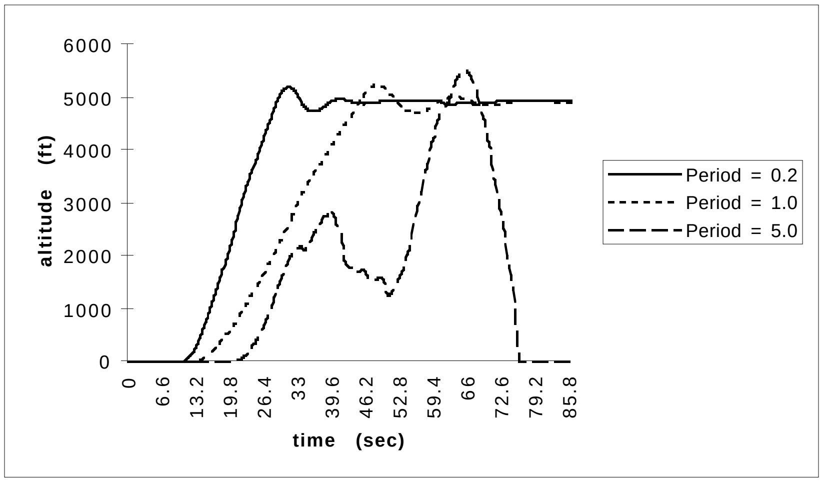


Figure 6-12: Aircraft Altitude for Varied *Ctrl* QoS Levels.

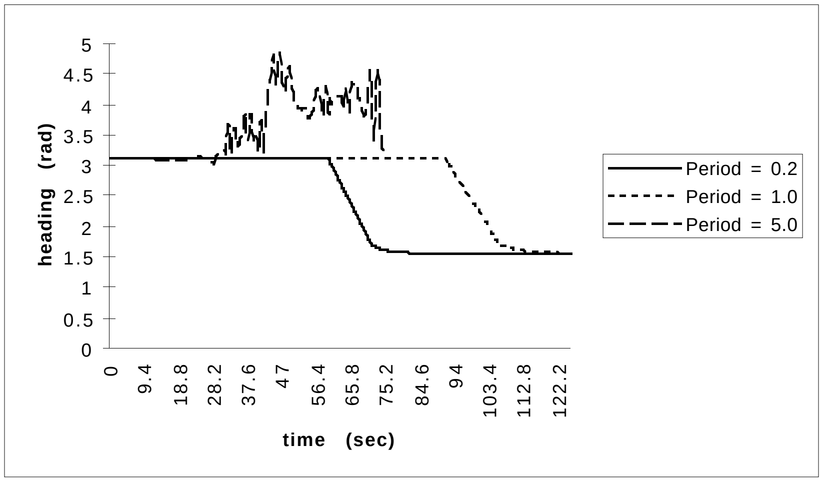


Figure 6-13: Aircraft Heading for Varied *Ctrl* QoS Levels.

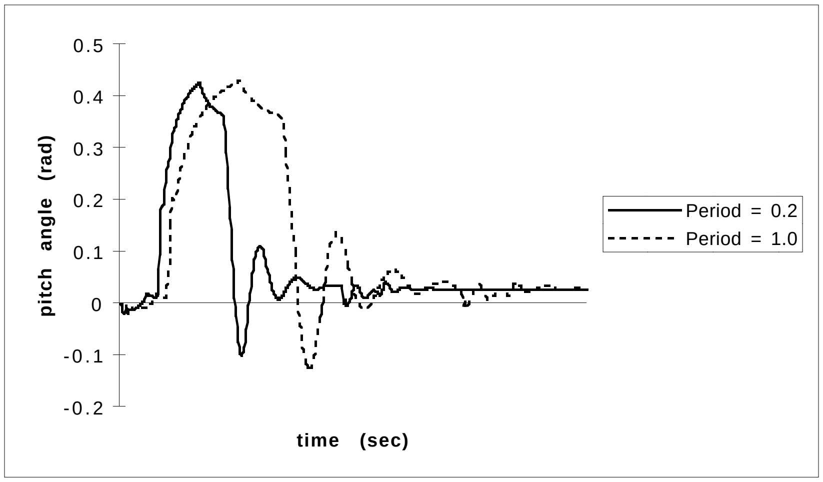


Figure 6-14: Aircraft Pitch Angle for Varied *Ctrl* QoS Levels.

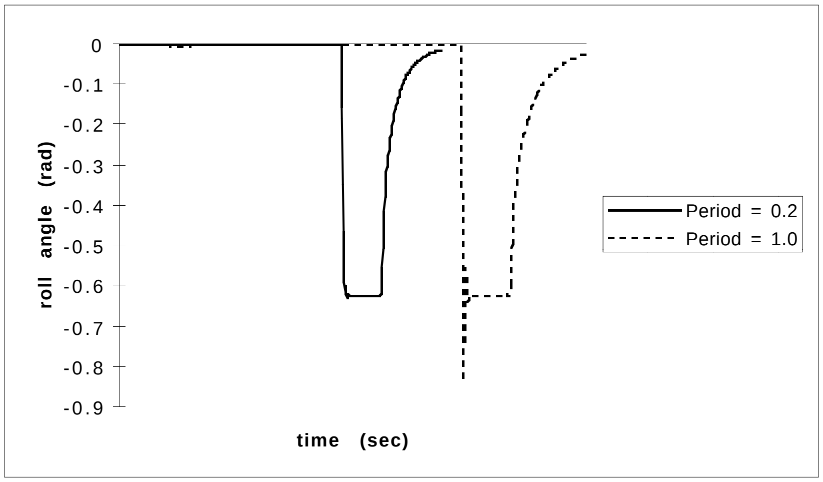


Figure 6-15: Aircraft Roll Angle for Varied *Ctrl* QoS Levels.

Through these examples, we have shown that our QoS negotiation scheme allows graceful performance degradation as resources become overloaded. It is important to note that, had we used traditional schedulability analysis algorithms that do not allow negotiated QoS degradation for this example, the system would have failed to guarantee/accept the entire task set on the same processor, leading to complete mission failure. We still have a substantial amount of CIRCA-II work to complete before a viable combination of dynamic scheduling and the existing "offline" scheduling algorithms can include QoS negotiation algorithms with or without the remainder of RTPOOL. However, given that in many circumstances the CIRCA-II planner may not otherwise produce a viable plan, the flexibility in task execution provided by QoS negotiation may be critical to mission success. We foresee such algorithms from RTPOOL and beyond as important future CIRCA-II additions to further enhance its ability to succeed in limited-resource systems that can experience computational resource faults/failures.

CHAPTER VII

FULLY-AUTOMATED FLIGHT WITH CIRCA-II

Automated aircraft flight is an attractive domain for this research because it requires a complex knowledge base and because continuous real-time control is essential since an aircraft cannot "stop and remain safe indefinitely" once it has left the ground. In today's commercial aircraft, flight management systems (FMS) [44],[64] are capable of automated flight from takeoff through landing, but only when the aircraft is operating within its nominal performance envelope (i.e., few, if any, anomalous situations have arisen). Mapped to CIRCA-II, the nominal goal achievement plans developed and cached offline are those that would correspond to FMS plans.

In this chapter, we focus on how CIRCA-II real-time failure avoidance methods can be used to enhance safety of automated flight during anomalous or emergency situations. As described previously, the first key is to identify each such situation, then quickly execute any required safety-preserving reaction. Additionally, since this reaction may make the aircraft deviate from its goal path, the CIRCA-II planner may be invoked during flight to produce plan(s) that will redirect the aircraft toward its goal, or toward a new goal (e.g., a safe landing as a minimum) when the original is unreachable.

We have developed a knowledge base to automate [simulated] aircraft flight and have tested it using a shareware aircraft simulator. Details of the simulator, CIRCA-II control of the aircraft, and results from simulated flights that include anomalous situations are described below. Following our initial tests with CIRCA-II, this simulator was adopted by researchers at Honeywell Technology Center (HTC) to demonstrate their research efforts and plans to DARPA (Defense Advanced Research Projects Agency). Recently, we demonstrated updated CIRCA-II capabilities as part of a joint research project with HTC. We highlight the safety-preserving activities of this Unmanned Combat Aerial Vehicle (UCAV) under CIRCA-II control as presented in this demonstration. Finally, we describe the University of Michigan's Uninhabited Aerial Vehicle (UAV) research project, originally conceived as a test platform for control researchers in the Aerospace Engineering department, then expanded to be a general testbed for autonomous flight experiments. Although the UAV is still in its initial test phase, we expect CIRCA-II will play a critical role in future UAV automation, and describe our ongoing efforts to apply CIRCA-II to flight planning, real-time process control, and fault recovery tasks.

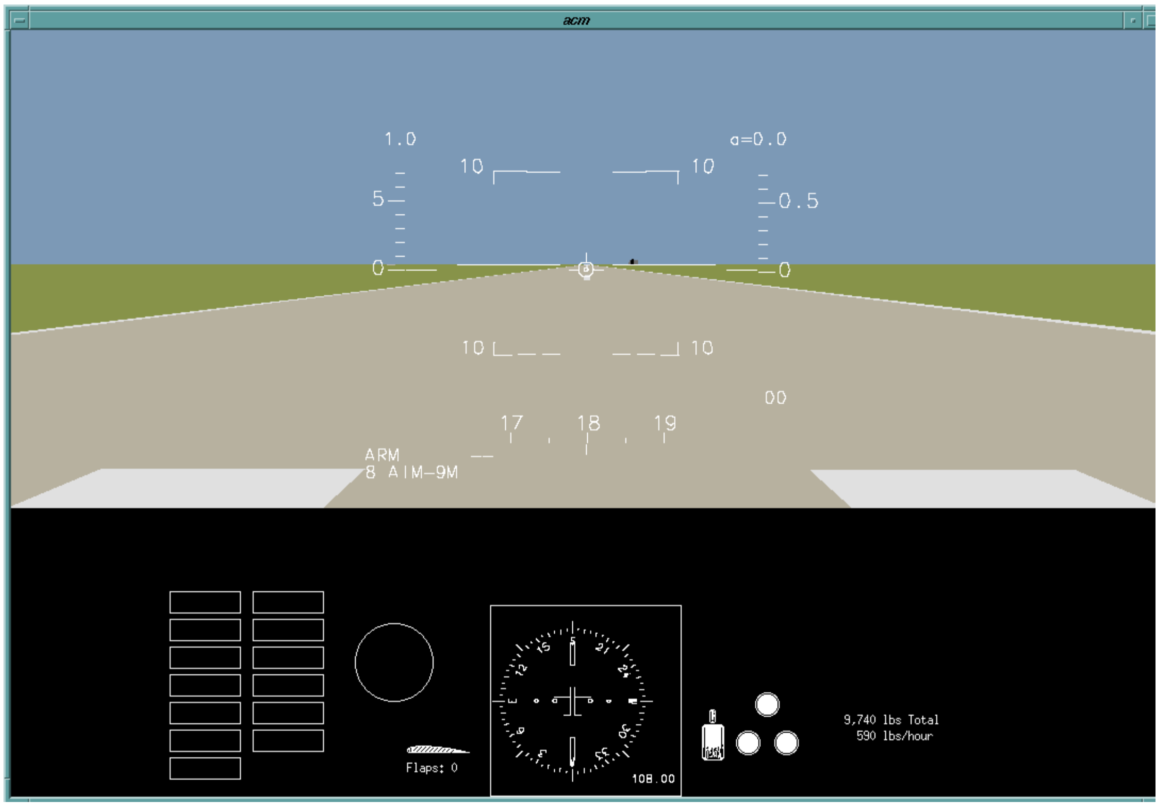


Figure 7-1: Aerial Combat (ACM) Flight Simulator: Cockpit Display.

Autonomous Flight on the ACM Simulator

The CIRCA-II algorithms have been applied to fully-automate the Aerial Combat (ACM) F-16 flight simulator [58], which we selected for our experiments due to its readily-available shareware source code and its realistic 6-degree-of-freedom model of flight dynamics. Figure 7-1 shows a cockpit view of the ACM simulation, originally built for human pilot training (or gaming) and adapted by us to allow fully-autonomous control using CIRCA-II. Because CIRCA-II is exclusively intended to perform high-level symbolic-based actions, we added a simple proportional-derivative aircraft controller set [60] linearized about two set points (i.e., elements of a gain-schedule table), one for the climb/cruise flight phases and one for final approach through landing.[59] CIRCA-II specifies altitude values as the reference input to the longitudinal controller and heading values as the lateral controller reference. For the remaining parameters required by the controllers (e.g., pitch angle or airspeed for the longitudinal controller), the controller contains internal default values used to compute actuator outputs.

Our CIRCA-II model is purposely designed to minimize the number of states required to complete the autonomous flight tests. Our model includes features for altitude, heading, location (relative to fixes in Figure 7-2), gear and traffic status, severe weather phenomena (severe turbulence, low-level wind-shear, tornado, etc.), and navigation sensor data, allowing a total of 50,176 modelable states. Appendix B includes a version of the ACM knowledge base we have used for CIRCA-II tests. A sequence of five subgoals have been constructed to achieve the "flight-around-the-pattern" trajectory illustrated by Figure 7-2. These subgoals include the takeoff climb and upwind flight to Fix1, the crosswind pattern leg to Fix2, downwind leg to Fix3, base leg to Fix4, then final

---

[59] The operating envelope for our controller is extremely limited but was sufficient for simulations of the gentle maneuvers performed by a commercial aircraft. The simulated F-16 is capable of performing far more aggressive maneuvers with a better controller.

approach to Fix0. Appendix B also shows the set of nominal plans generated by the CIRCA-II Planning Subsystem that successfully automate the flight-around-the-pattern task from takeoff through landing.[60] To generate this control plan set, CIRCA-II only required the generation of approximately 200 of the 50,000 states.

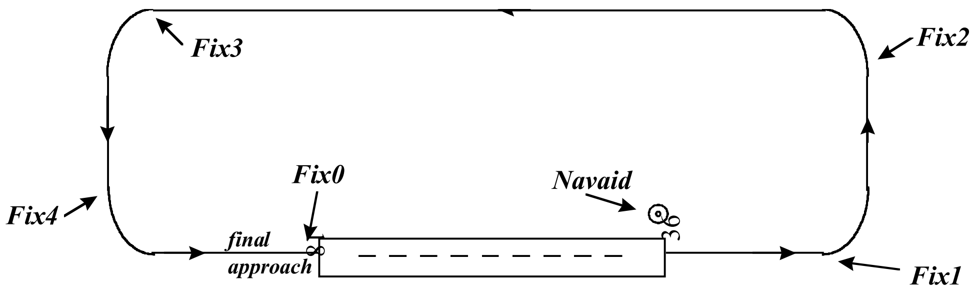


Figure 7-2: Simulated Flight Pattern.

Once the nominal flight was successfully automated, we introduced two emergencies: “gear fails on final approach”, and “collision-course traffic”. If either situation occurs, failure to notice and react to the problem results in an aircraft crash (modeled for CIRCA-II as *ttfs*). Our knowledge base contains a "gear" feature with values "up" and "down", and a "traffic" feature with values "true" and "false". We also include several features (see Appendix B) for swerving to avoid traffic then resuming the designated flight path to the next pattern Fix. We assign simplistic probability rate functions for the various temporal transitions, predominantly a constant rate value with zero probability below *min*Δ for *ttfs* and a probability 1 at *max*Δ (then zero above) for reliable *tts*.[61]

---

[60] In simulation, the presence of anomalous events was explicitly controlled by the user. This greatly facilitated initial debugging of the basic "flight-around-the-pattern" activity.

[61] The probability functions assigned to all examples shown in this chapter are not based on actual statistics, but instead are intended to demonstrate CIRCA-II functionality. Gathering accurate statistics for in-flight events was well beyond the scope of this dissertation.

During normal flight, the gear is down and no collision-course traffic is present. A *tt* to gear indicating “up” has a very small constant probability rate, while a *tt* to detecting collision-course traffic is also unlikely but has a higher probability rate than the "gear up" *tt*. Also, a temporal transition to failure (*ttf*) is included to capture the scenario of landing with gear up, which inevitably results in a crash. If it should occur, the proper "pilot" reaction to a gear failure on final approach is to execute a go-around (i.e., continue around the pattern a second time), performing available actions such as cycling the gear retract/extend mechanism. We do not model the gear cycling activity in our knowledge base, but the CIRCA-II planner naturally designs the "go-around" into any plan that must handle gear failure since the planner reasons that any landing while the gear is up will result in a *ttf*.

We purposely increased action worst-case execution times to test CIRCA-II's ability to make tradeoffs when resources were over-utilized. During plan development, the scheduler is then unable to guarantee all failure-avoidance activities for the "final approach" plan but determines it can schedule all activities except reaction to the [low-probability] gear failure. The scheduler recommends a probability threshold $P_{removed}$ to the planner which results in the removal of all states with "gear up". The nominal plan developed next is schedulable but contains no actions for handling failed gear, except for a "removed state" detection TAP activated when gear fails (see "final approach" plan in Appendix B). The planner then develops a contingency plan that effectively performs a go-around, leading the aircraft back on a course to "Fix1" (see Figure 7-2). This contingency plan is also shown in Appendix B.

To illustrate CIRCA-II's ability to succeed with incomplete knowledge, we removed the gear failure *tt* from the knowledge base and re-ran CIRCA-II. For this final approach plan, CIRCA-II never expands any "gear up" states as part of a nominal plan because no *tt* leads to this situation. However, an imminent-failure detection TAP is built into the plan since the *ttf* to a crash when landing gear-up is modeled. And, because

CIRCA-II also builds contingency plans for imminent-failure states, a "gear up" reactive plan is again built and stored in the plan cache, resulting in an identical response during plan-execution to that developed when the "gear up" states were removed. The only difference in this case is that CIRCA-II had no knowledge of that transition in this second test run.

During execution, if the gear fails at any point during the climb and cruise flight phases (i.e., plan0-plan3 in Appendix B), the executing CIRCA-II plans will not notice, because no failure is imminent and the goals of getting to the next pattern “FIX” can still be achieved.[62] However, if the gear has either failed before initiating the approach or during the approach (so long as it doesn’t fail at touchdown), the failure is detected, the cached "go-around" plan is initiated to “buy time”, and the CIRCA-II planner is notified of the switch to a contingency plan. The CIRCA-II planner replans for the next achievable subgoal, effectively re-directing the plane around the pattern a second time. This allows re-use of all subsequent cached plans *(plan1-plan4)* for continued flight around the pattern.[63] If the gear fortuitously extend during the second flight around the pattern, the aircraft will land safely. However, if the gear has failed permanently, CIRCA-II continues executing go-arounds indefinitely, unaware that a crash is inevitable when the *unmodeled* “run-out-of-fuel” transition occurs.

The ACM simulation continues to be utilized for testing basic algorithms as they are implemented in CIRCA-II. However, we have more recently become involved with

---

[62] This behavior illustrates how CIRCA-II can save plan space and execution time by not considering “unnecessary” events; however, this behavior also illustrates how a system cannot simply detect when it is in danger without explicitly adding this to the set of planned actions.

[63] As future work, we would like the planner to explicitly reason about the available cached plans to direct the system back toward one of these plans when possible. For this flight example, we are simply lucky that the exact goal states match both times around the pattern.

two other aircraft-related projects. Below, we describe tests with Honeywell's UCAV simulation. Then, we venture into the "real-world" of actual aircraft flight in which we hope to better-validate CIRCA-II algorithms in ongoing work.

Demonstration of CIRCA-II on the ACM/Honeywell UCAV

The ACM simulator described above has been adopted by Honeywell Technology Center and augmented to perform as an Unmanned Combat Aerial Vehicle (UCAV) simulation platform. Although the basic dynamics, control laws, and CIRCA-II interface remain the same, the role of CIRCA-II has changed from responding to aircraft anomalies to maneuvering away from attacking missiles while attacking targets. Figure 7-3 illustrates an overall view of the world as seen by the UCAV.

In previous simulations, CIRCA-II plans were responsible for directing the aircraft away from dangerous situations as well as dictating the waypoint trajectory followed during flight. In the UCAV simulation, we rely on a separate trajectory generator to specify aircraft waypoints, then use CIRCA-II as it was intended: to avoid catastrophic failure situations. For the UCAV, a failure translates to the aircraft being hit by an enemy missile. Thus, our CIRCA-II knowledge focuses on modeling missile attack scenarios and reacting appropriately to them.

In our simple model developed for a recent coordinated demonstration with Honeywell, we modeled two classes of weapons that may attack the UCAV: *radar missiles* that typically attack from high altitude and *infrared (IR) missiles* that can only detect and attack the aircraft when it is near the ground. Appendix C shows the knowledge base we used during this demonstration. This knowledge base contains the temporal transition probability rate functions illustrated in Figure 7-4.

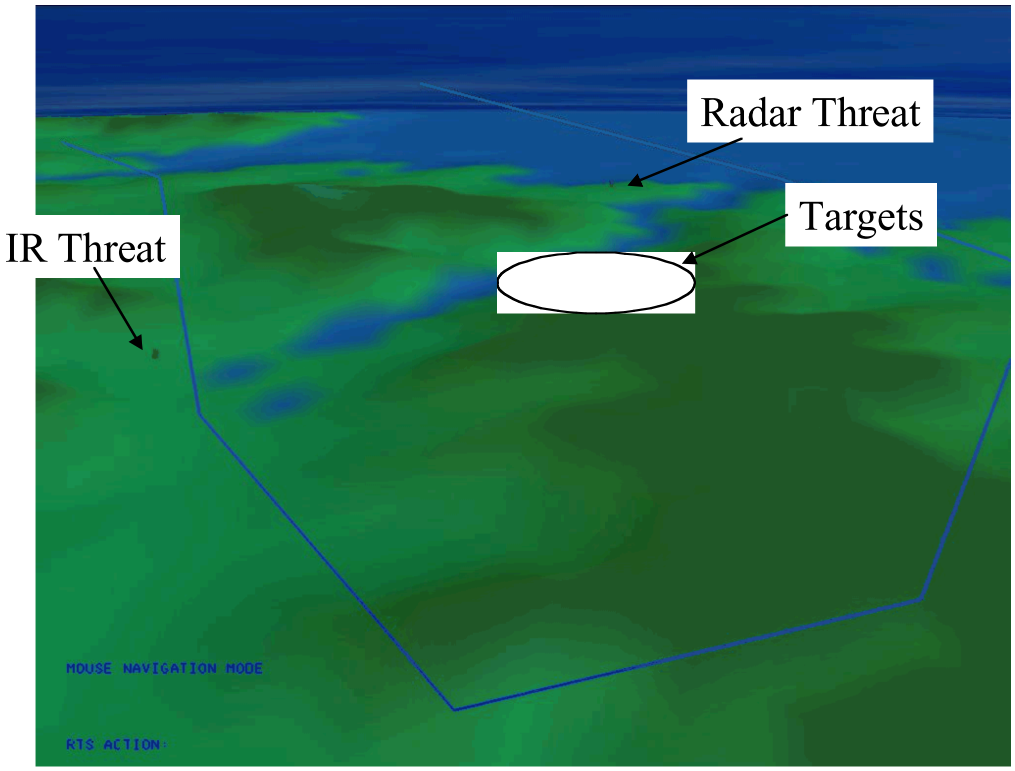


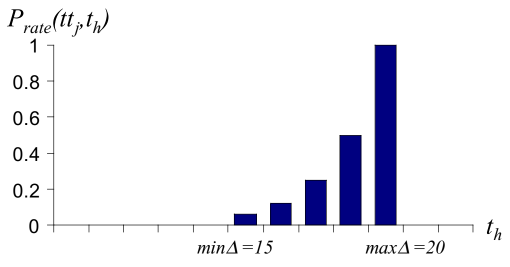
Figure 7-3: Hostile Environment Encountered during UCAV Flight.

To keep our model and state-space relatively simple for illustrative purposes, we model only two *ttfs*, *radar_kill* and *IR_kill*. Each of these *ttfs* has a non-trivial delay before it can occur, as labeled by *minΔ* in Figure 7-4. Then, for each time step, *IR_kill* has a higher probability than *radar_kill*. For the CIRCA-II demonstration, we carefully engineered the knowledge base so that we cannot schedule all actions required to avoid both *ttfs* in one plan. Then, we ran a series of simulations to illustrate how CIRCA-II still is able to minimize the likelihood of all *ttfs*.

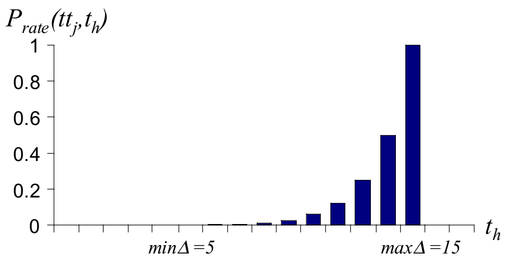
For the first test run, we presume no probabilistic guarantee tradeoff mechanism during CIRCA-II plan development. In this case, no plan could be scheduled to evade both radar and IR missiles. However, we acknowledge that a higher-level decision maker can easily pick the missile encounter with the lowest probability to ignore. In this case, since *radar_threat* has the lower P_{rate} magnitude, it is removed from the knowledge base and a plan is built without further difficulty. This plan is shown in Figure 7-5. During the simulated flight, the aircraft successfully avoids any IR missiles encountered, but does not even attempt to evade radar missiles, thus they kill the aircraft when launched.

Because the majority of UCAV missions are flown at relatively high altitude, the probability of being in a low-altitude state is low as modeled by the low-probability *swoop tt*. For our next test, we reinstated the complete radar and IR missile transition model and allowed CIRCA-II to build a plan that exhibited probabilistic guarantees. However, we disabled the unhandled state detection and reaction software, thereby forcing CIRCA-II to utilize only one plan during the entire flight. For this scenario, CIRCA-II uses the state-probability computation model described in Chapter IV to compute that encountering a radar missile is actually much more likely than encountering an IR missile (even though the IR_threat has higher P_{rate}) because the probability of the aircraft being in a "high altitude" state is much greater than being in a "low altitude" state where the *IR_threat* may occur. Figure 7-6 shows the state-space diagram for this scenario. During flight, this plan allows the aircraft to respond to all radar missiles but the aircraft fails to evade all IR missiles. Given the primarily-high-altitude mission, this scenario improves the likelihood of survival over the previous test, but the aircraft still can be killed when encountering an IR missile.

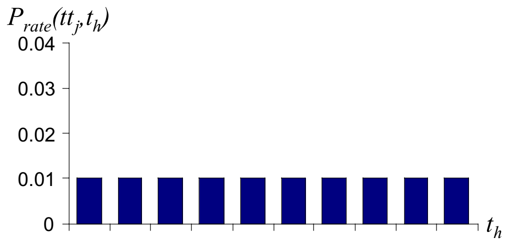
For the final demonstration, we reinstated unhandled state detection algorithms and directed CIRCA-II to construct and cache both nominal and contingency plans. Figure 7-7 shows the state-space expanded for both the nominal and contingency plans used by the CIRCA-II Plan Executor during flight. The nominal plan is identical to that constructed in the previous experiment. However, now when an IR_threat is encountered, a "removed state" is flagged and the dispatcher switches to the cached contingency plan to handle the threat. After the threat has passed, the dispatcher switches back to the nominal plan which executes until either another IR threat is encountered or else the mission terminates as illustrated by the single absorbing state with state feature *Path=Done*.



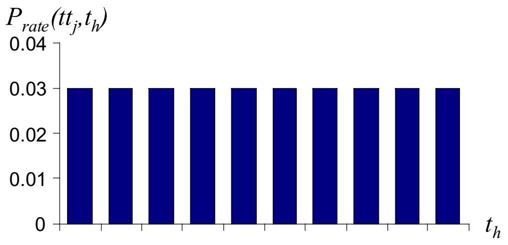
a) *evade_radar_missile*



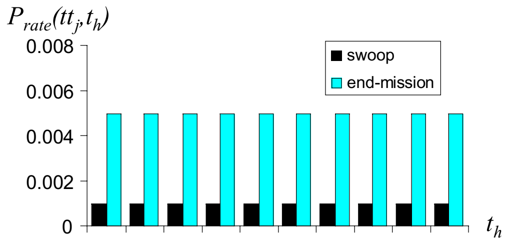
b) *evade_IR_missile*



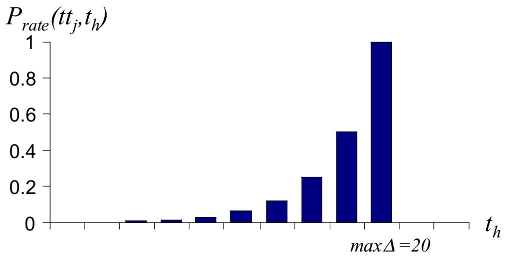
c) *Radar_threat*



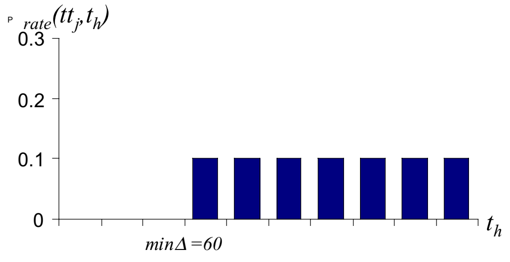
d) *IR_threat*



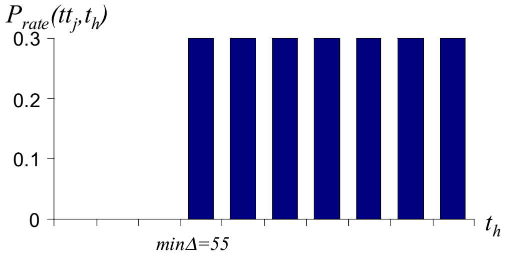
e) *swoop* and *end-mission*



f) *climb*



g) *radar_kill*



h) *IR_kill*

Figure 7-4: UCAV *tt* Probability Rate Functions.

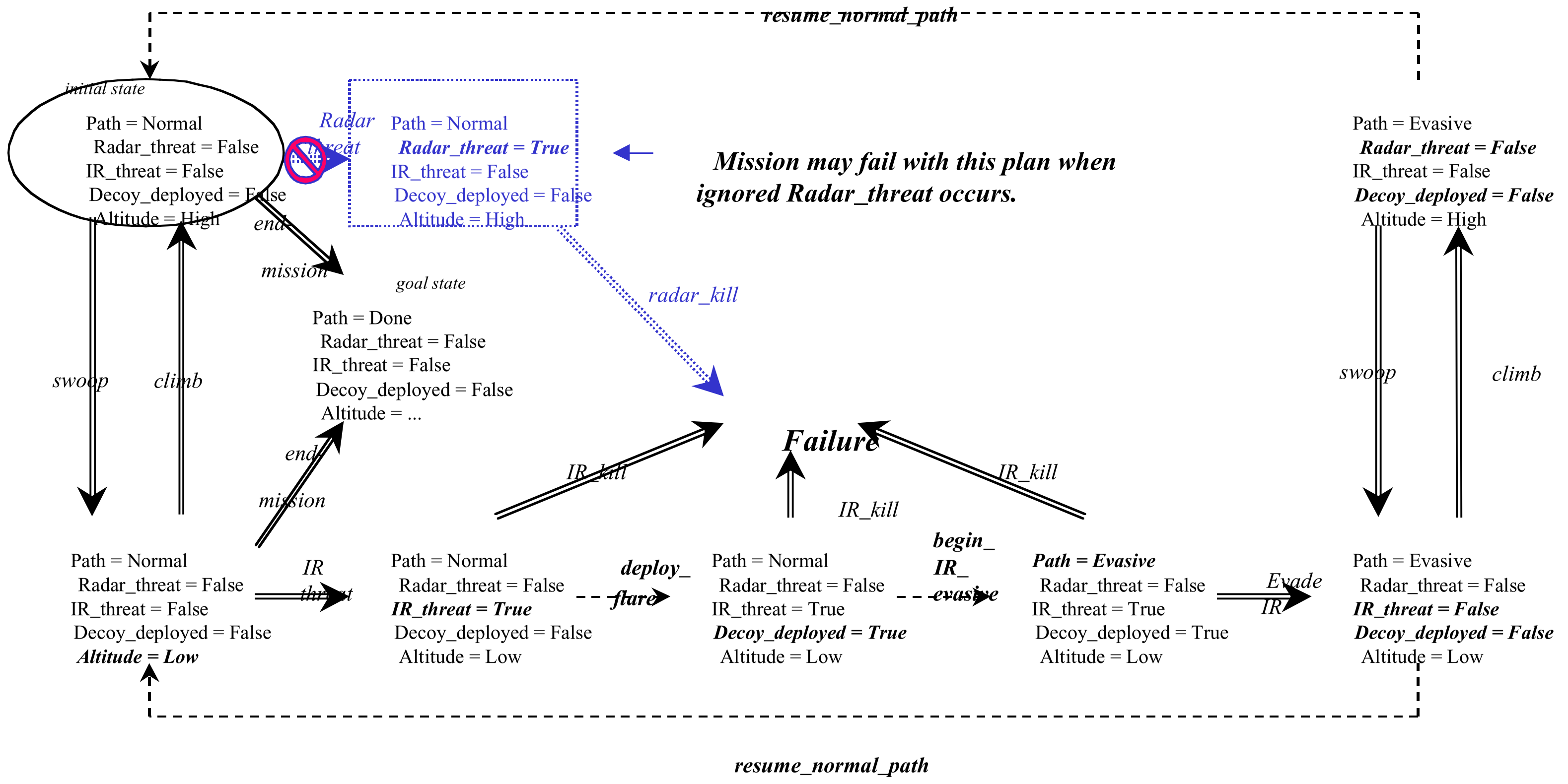


Figure 7-5: UCAV State-Space with no Radar-threat Model.

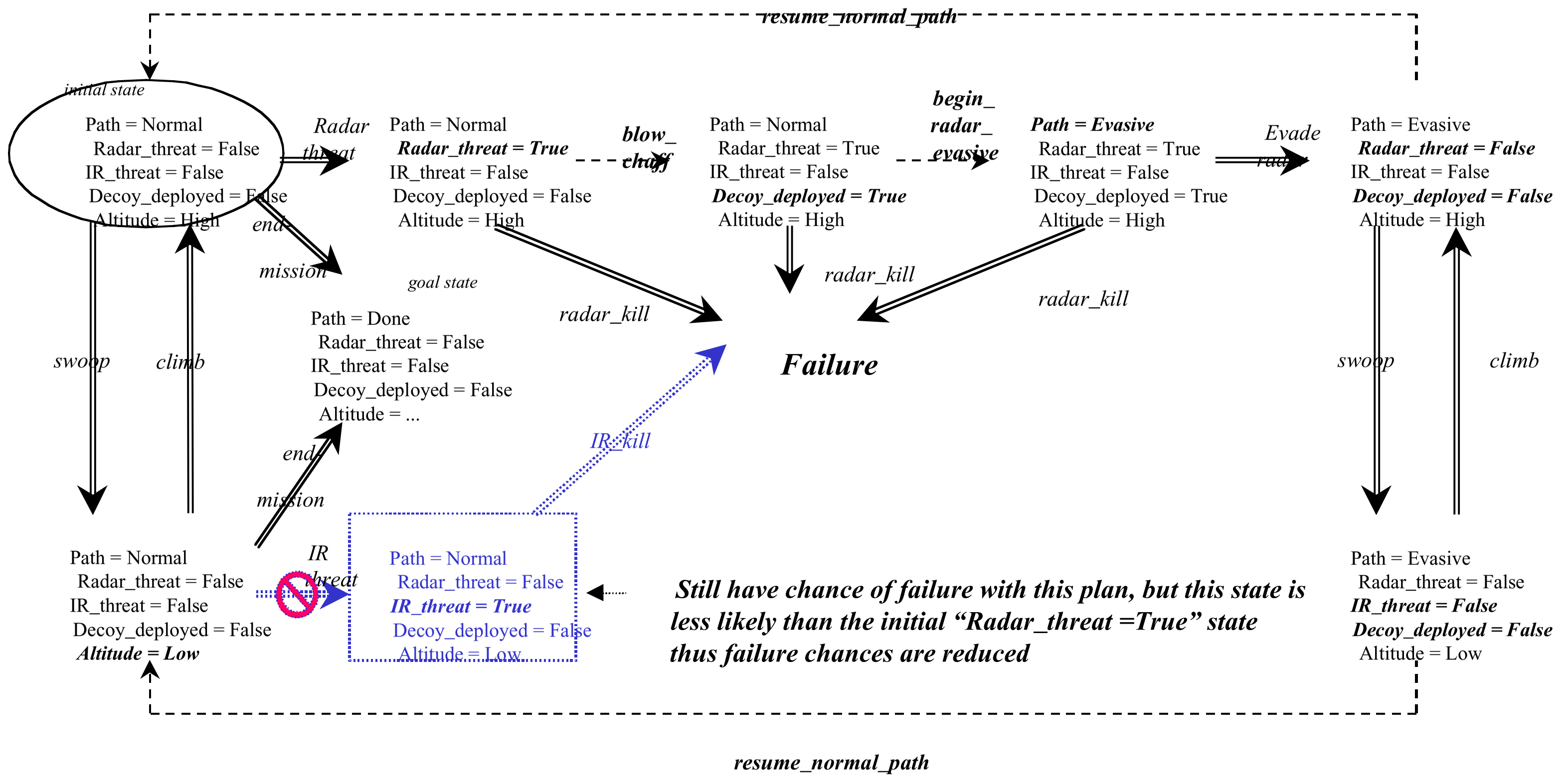


Figure 7-6: UCAV State-space with Radar-threat -- Optimize Tradeoffs in Nominal Plan.

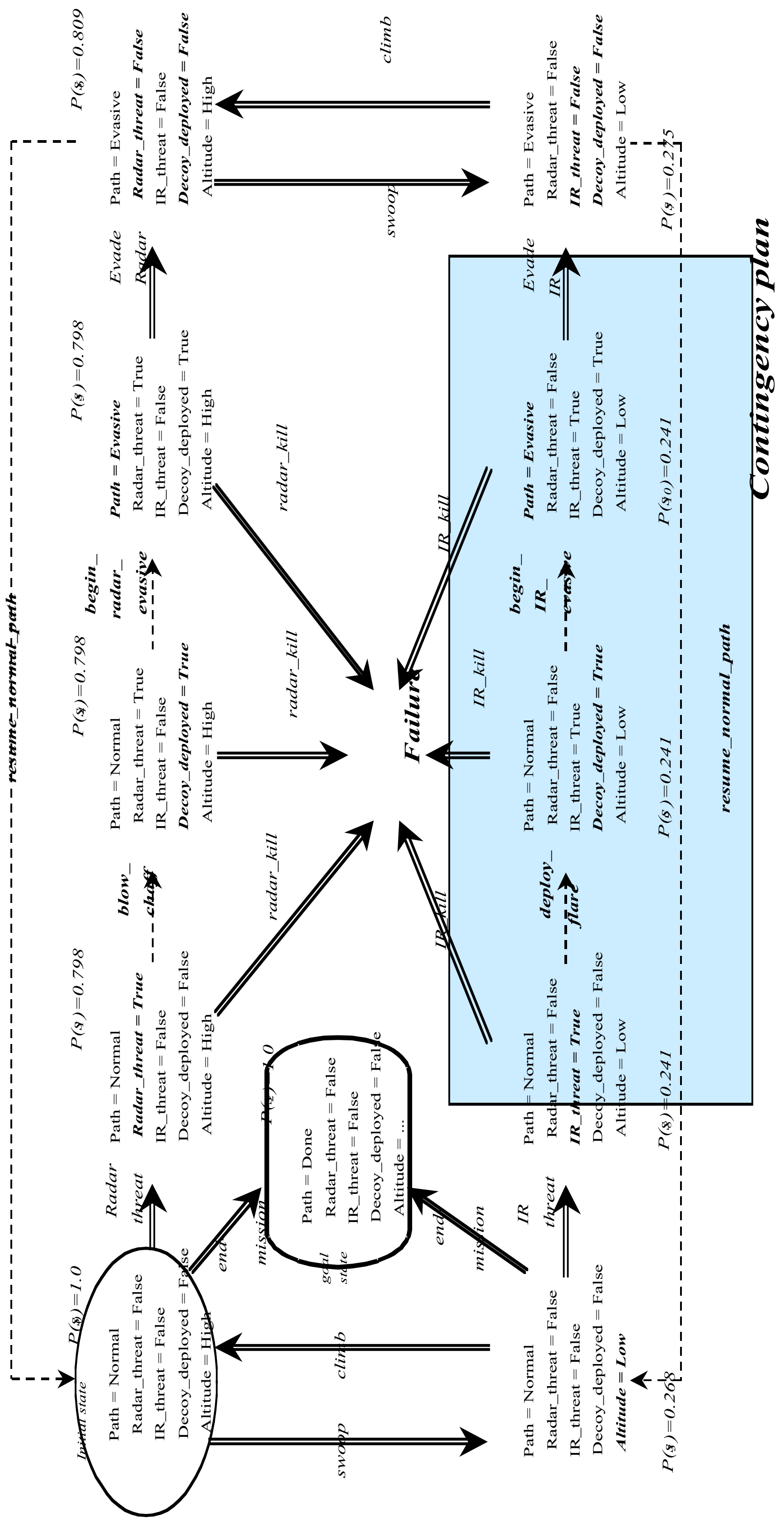


Figure 7-7: UCAV State-space with Radar-threat -- Nominal+Contingency Plans.

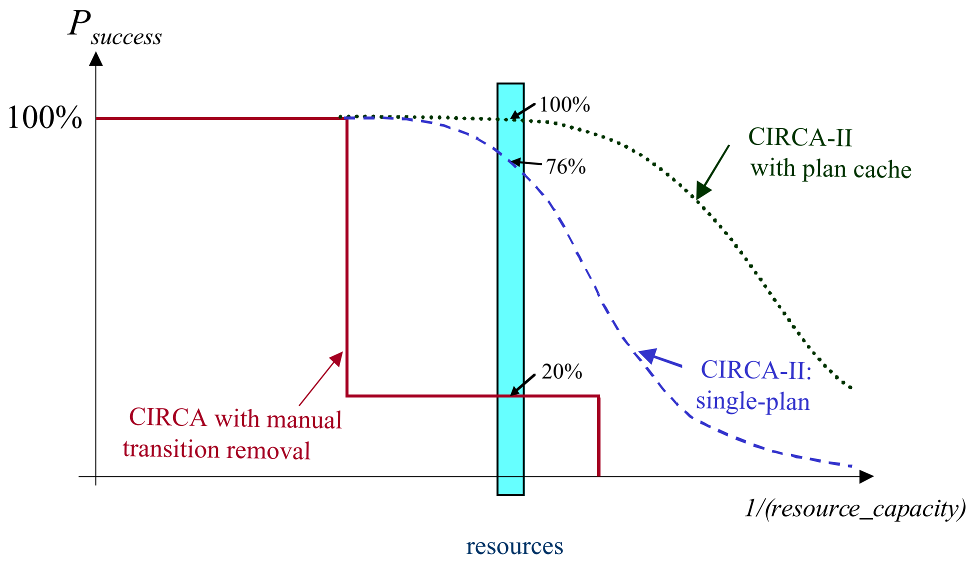


Figure 7-8: Success Probability vs. Resource Capacity: UCAV Flight Test.

Figure 7-8 revisits the success vs. resource capacity curve illustrated in previous chapters, this time focusing on the specific UCAV test series described above. Using the state probabilities computed by CIRCA-II (shown in Figure 7-7), we observe that if the *radar_threat* transition is ignored, the probability of success is only 20% (presuming an ignored radar threat always leads to failure) since the probability of visiting the first state along the *radar_threat* path is 80%. When an *IR-threat* is ignored, success probability jumps to 76% because of the relatively low probability (24%) of actually visiting the path along which an *IR_threat* has occurred. Finally, the probability of success jumps back to 100% when all threats are handled within the real-time plan set.

This series of UCAV tests has been used to demonstrate that CIRCA-II increases system robustness by using probabilistic guarantees to make intelligent tradeoffs about what to ignore then detecting and responding to all ignored states. In the first test, we use an *ad hoc* method for choosing a state transition to remove from the knowledge base. This leads to a plan which had a relatively low probability of success since radar missiles are ignored but are also most likely to be encountered. The intermediate experiment increases the chances of success by handling the more-likely radar missiles, but still ignores IR threats. The final test illustrates how CIRCA-II can be used to handle more situations than can be scheduled into a single plan, thereby allowing the aircraft to respond to both radar and IR threats when they occur. Although these tests are relatively simple, we expect similar results with an even more constrained model until the plan retrieval process simply cannot occur quickly enough to avoid failure modes.

Flight with the University of Michigan UAV

The University of Michigan has recently embarked on an extensive Uninhabited Aerial Vehicle (UAV) project, cooperatively supported by Michigan's Aerospace and EECS departments. The purpose of this section is to illustrate how CIRCA-II will be applied to the task of fully-automating a "real" aircraft, as opposed to the carefully-constructed simulation tests in which we had complete control over anomalous events. We begin this section with a description of the UAV, its onboard instrumentation, and the software architecture which is designed to achieve the real-time behavior required for reliable automatic control. We also describe the integral role CIRCA-II plays in enabling this architecture to support the fully-automated flight configuration.

We originally planned to include UAV test flight data in this dissertation to further illustrate the utility of CIRCA-II for maintaining safety in a real-world system. However, unforeseen difficulties with UAV hardware, including electrical noise and power system failures, have delayed test flights to the extent that we have been unable to include actual UAV data for CIRCA-controlled flights in this dissertation. Currently, we are in the process of completing sensor calibration flights and within the next month hope to begin incorporating aircraft dynamic parameters into our state estimation software. Then, we must implement and test a minimally-capable longitudinal controller for straight-line flight with gentle climbs/descents, after which we will couple this controller to a lateral controller to allow gentle-bank turns to any commanded heading. Because the high-level actions executed by CIRCA-II can only be utilized after incorporation of minimally-operational state estimation, control, and guidance software, CIRCA-II tests on the UAV must be delayed past the verification of this low-level control software. However, even with such delays, we have forged ahead with development of a preliminary abstract plan set we hope CIRCA-II will generate and test on the UAV within the next year. We conclude this section with a high-level description of these plans.

Figure 7-9 shows a picture of the University of Michigan UAV, a radio-controlled (R/C) pusher-prop airplane with eleven foot wing span and a gross weight of just under 55 pounds.[64] The aircraft accommodates all sensors required for full automation, including air-data system (to measure airspeed, angle-of-attack, and sideslip-angle), differential GPS, inertial measurement unit (to measure aircraft body-axis angular velocities and accelerations), tachometer, and complete set of potentiometers to measure control surface deflection. Additionally, the current UAV is explicitly over-designed to be very stable during normal flight, facilitating both controller design and manual flight by inexperienced R/C pilots (including the author of this dissertation).

The UAV is flown using both onboard and ground-based computers, connected as shown in Figure 7-10. Sensors are filtered and read by onboard processor **P1**, and actuator commands are transmitted from this same processor**.** Onboard processor **P2** reads positions from the d-GPS unit, and communicates via a real-time serial link to the ground station computer (**G1**). A human pilot maintains override capability via a standard Pulse Position Modulation (PPM) R/C transmitter/receiver pair, as the "ultimate" real-time failure avoidance mechanism for our research flights.

[64] Our original UAV was an off-the-shelf Citabria model, shown in [6]. After adding the minimal instrumentation required for automation, the flight-worthiness of the Citabria was roughly equivalent to that of a radio-controlled brontosaurus. As a result, we designed the current aircraft specifically to carry and easily access this instrumentation.

All UAV processors (**P1, P2**, and **G1**) run the QNX real-time operating system, so that hard real-time tasks can have reliable execution guarantees. Figure 7-11 shows the high-level process set run on the UAV processors. Processor **P1** runs exclusively hard real-time tasks at frequencies ranging from 10 Hz (Controller) to 100 Hz (Sensor sampling). Task structure and worst-case execution properties can be characterized in advance for the P1 processes regardless of the aircraft's specific mission plan. So, hard real-time execution on **P1** is guaranteed from a fixed schedule.[65]

P2 contains tasks executing with longer deadlines (on the order of seconds), but most of these tasks have larger execution variance than those on **P1**. Model identification (ID), guidance, and serial communications may have near-constant execution times during "nominal" flight cases (e.g., model requires minimal modification, a "regular" set of status messages is sent to G1). However, during anomalous situations (e.g., requiring a new dynamic parameter estimate), adaptive algorithms such as that used for model ID [35] may require more resources to converge upon an accurate solution. Additionally, higher-level mission-related tasks, like selecting the next waypoint in the trajectory, will require more resources, particularly during anomalous situations for which the flight plan must be altered. The ground station computer (G1), running solely in a *soft* real-time execution mode, includes a real-time serial server for low-bandwidth communications with the aircraft and a GUI for researcher observation of flight status.

As shown in Figure 7-11, CIRCA-II is an integral part of the UAV software architecture. Before flight, the Planning Subsystem develops and uplinks to the aircraft the set of nominal flight mission plans as well as a set of contingency plans to handle improbable emergency situations. All these plans are sent via the serial link to the Plan-Execution Subsystem on aircraft processor **P2**, which then begins executing the first

[65] Additionally, the **P1** processes have a low-variance in execution times, so we expect the worst-case to be near the average-case, thus **P1** will typically have little slack time.

nominal mission plan. CIRCA-II plans for the UAV include actions to specify the high-level waypoint trajectory (analogous to the pattern fix model used in simulation tests) and transmit results to the guidance process. Guidance is responsible for continuous reference signal generation of altitude, airspeed, and heading used by the controller.

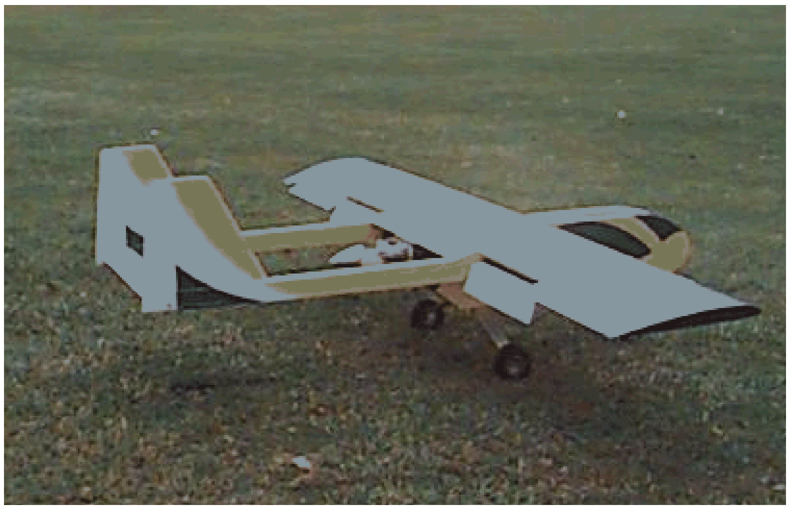


Figure 7-9: University of Michigan UAV.

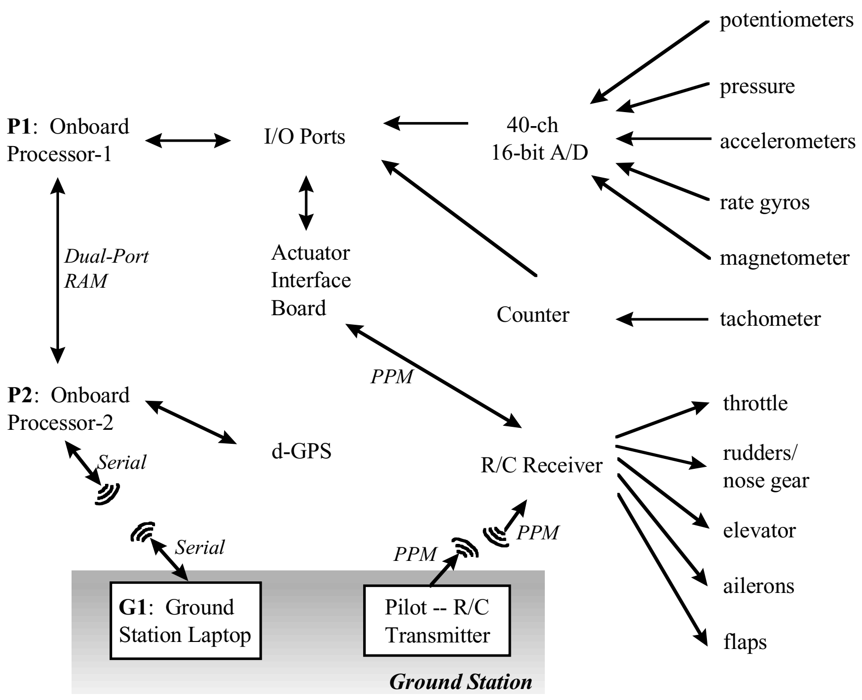


Figure 7-10: UAV Data Acquisition and Communication Systems.

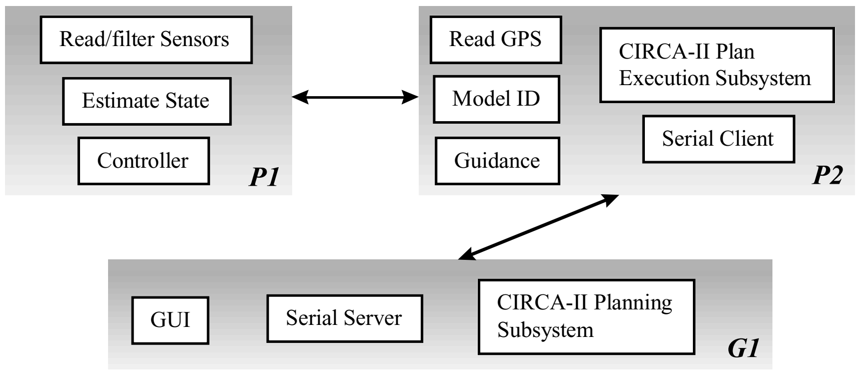


Figure 7-11: UAV Software Architecture.

CIRCA-II will be responsible for controlling execution of all **P2** processes except reading GPS, effectively treating guidance, ID, and the serial client processes as TAP actions which must have predictable worst-case execution properties, or else be interruptible should the allotted worst-case execution time expire. CIRCA-II feature values for TAP tests will be derived from the state estimate, available to all **P2** processes via the dual-port memory connection to **P1**. In this manner, feature "tests" will effectively be instantaneous, and the worst-case execution times for all CIRCA-II TAPs will be solely due to actions.

Although we have no concrete UAV dynamic model or control software to utilize for CIRCA-II flight tests, we have begun to develop a CIRCA-II knowledge base and plan set we believe is feasible given the predicted UAV model properties. In conjunction with control researchers in the Aerospace Engineering Department, we plan to study our UAV's ability to perform fault detection, identification, and recovery to two specific emergency situations: *engine failure* (presuming a single-engine aircraft like our UAV) and *airframe icing*.[66] We selected these particular emergencies because they are the most

[66] During UAV flight tests, engine-failure is easy to simulate by setting the throttle to idle. We plan to simulate airframe icing with software between controller and actuator command output that reduces the magnitude of control surface commands in accordance with the expected reduction for the [simulated] amount of ice present.

common occurrences that are also not adequately detected or handled by existing flight management systems, and that are interesting both from the control and high-level mission reconfiguration (i.e., contingency response) perspective. From the full UAV software architecture perspective, "detection and isolation" of an engine failure or icing event will be performed by the model identification software. Initial research efforts toward the detection and isolation of airframe icing based on actual test data from a twin-otter research aircraft are described in [48] and [49]. CIRCA-II then interfaces to the ID module with feature flags that indicate whether an icing or engine-failure event has occurred *((Icing=True) or (Engine_failure=True))*, and must plan explicit reactions to respond appropriately to these events.

Figure 7-12 describes a “nominal” flight plan for UAV flight around the R/C airfield, similar to the pattern shown in Figure 7-2. We include TAPs that allow updating and following the flight trajectory around the flight pattern, as well as TAPs to detect the improbable *engine-failure* and *airframe-icing* events. Because a significant fraction of **P2** resources may be required to respond to either of these improbable events, we expect that *engine-failure* and *airframe-icing* will each be handled with a contingency plan. Figures 7-13 and 7-14 outline the contingency plans required to avoid failure after an *engine-failure* or *airframe-icing* event, respectively. As illustrated by these plans, goal-achievement actions (e.g., go around the complete rectangular pattern as in Figure 7-2) are not included. Instead, the safety of the UAV is considered in terms of the specific emergency encountered. To summarize, the *engine-failure* contingency plan directs the system to land anywhere on the R/C field when possible or in a straight-ahead location when required by low altitude (instead of attempting the classic "impossible turn" back to the field that would result in a stall-and-crash situation). The “airframe icing” contingency response focuses on exiting the [simulated] clouds, and updating weather as permissible during slack intervals between stability-preserving actions.

```
Tap 1:  If (altitude or course error)
             Update dynamic UAV model                 (guaranteed)
Tap 2:  If (engine-failure or airframe-icing)
             Notify plan dispatcher of unhandled state (guaranteed)
Tap 3:  If (approaching pattern corner)
             Turn left to next heading                (best-effort)
Tap 4:  If (on final approach to landing)
             Notify plan dispatcher of goal state     (best-effort)
```

Figure 7-12: Overview of the "Nominal" CIRCA-II UAV Flight Plan.

```
Tap 1:  If (approaching stall)
             Establish best-glide via model update.   (guaranteed)
Tap 2:  If (altitude < minimum)
             Setup for landing ahead.                 (guaranteed)
Tap 3:  If (minimum ≤ altitude)
             Set turn time/direction so UAV follows best glide-path
             to R/C field.                            (guaranteed)
Tap 4:  If (True)
             Attempt engine restart.                  (best-effort)
Tap 5:  If (Engine restarted)
             Notify plan dispatcher of unhandled state (best-effort)
Tap 6:  If (No code sent yet)
             Transmit emergency code.                 (best-effort)
```

Figure 7-13: Overview of the CIRCA-II "Engine-failure" Contingency Plan.

```
Tap 1:  If (approaching stall)
             Update dynamic model to fit icing parameters.  (guaranteed)
Tap 2:  If (just entered clouds or precipitation and no previous icing)
             Turn 180° to get out of clouds.                (guaranteed)
Tap 3:  If (aircraft can climb and near cloud tops)
             Climb out of ice.                              (guaranteed)
Tap 4:  If (not near terrain and icing prohibits climb
            and surrounded by clouds)
             Descend to melt ice.                           (guaranteed)
Tap 5:  If (airframe icing dissipated)
             Notify plan dispatcher of unhandled state     (best-effort)
Tap 6:  If (communications operational)
             Report icing and get updated weather report.  (best-effort)
```

Figure 7-14: Overview of the CIRCA-II "Airframe-icing" Contingency Plan.

We have not yet determined how to build the CIRCA-II knowledge base so that it will generate the proposed best-effort actions for these contingency plans, but include them as part of our abstract plans in this section because our ultimate goal is to allow CIRCA-II to at least emulate human pilot activities in emergency situations. We suspect that as automatic subgoaling is better developed in the CIRCA-II architecture, we can utilize this capability to identify and handle reduced goals (e.g., notify air traffic control of the impending emergency landing when the engine fails) in contingency plans instead of planning for *no* particular goals as is currently the default for contingency plan construction.

In this chapter, we have described the progression of CIRCA-II tests from successfully automating simulated flight-around-the-pattern to automating a simulated UCAV mission with multiple threats. Although no concrete results from actual UAV flights are yet available, we expect to continue the pursuit of integrating CIRCA-II into the UAV control system to maximize UAV safety via operation only in the controllable partition of its state-space even when significant emergency situations are encountered.

CHAPTER VIII
CONCLUSION

This dissertation introduced techniques that permit structured tradeoffs in an integrated plan generation and execution system when computational resources are limited, domain knowledge is uncertain and may also be imprecise, and dynamic world events occur rapidly. In accordance with an intuitive "safety first" policy, we required guaranteed hard real-time response to all dangerous world states but allowed best-effort reactions in states where the action strictly enhances mission goal-achievement. This prioritization was originally introduced in the Cooperative Intelligent Real-time Control Architecture (CIRCA) [51], which was designed to build control plans that are explicitly scheduled to provide hard real-time failure-avoidance guarantees when executed. We adopted CIRCA as the basis for this research and described its evolution to CIRCA-II.

The purpose of the planning system in CIRCA-II is to create control plans that guarantee safety while attempting to reach mission goals when executed. It is unrealistic to assume that an absolutely safe plan will always fit on a limited-resource execution platform, so we have incorporated the necessary capabilities that allow the CIRCA-II planner to degrade safety guarantees from absolute to *probabilistic* when required. To this end, we have implemented a stochastic state-space planner that handles uncertain domain knowledge and prioritizes the reachable state-space based on the relative likelihood of visiting each state. Our planner expands states using a knowledge base specified by a transition set with unconditional time-dependent probabilities, and employs a variety of weighted-average functions to approximate the likelihood of ever visiting each reachable state, even when the state-space is cyclic and dependent temporal transition chains (see Chapter IV) are present. The plans developed with our model are sufficient for failure-avoidance purposes, but may not be optimal, particularly with respect to goal achievement. We considered a Markov Decision Process (MDP) planner [11] as a particularly attractive alternative approach because of its ability to generate optimal policies. However, due to the additional complexity required for an MDP planner to represent and reason about action deadlines and state "history", we opted to use our simpler but approximate model. If we utilize state probabilities for more than relative state prioritization in future work, we will revisit the possibility of incorporating an MDP planning model into our architecture.

After prioritizing the state-space in terms of probabilities, we were able to relax safety guarantees from absolute to probabilistic by constructing plans that only classify a state as reachable if the probability of visiting that state is above a flexible threshold $P_{removed}$. In the current CIRCA-II implementation, a value for this threshold was recommended by the scheduler from an analysis of proposed task priorities, probabilities, and identified bottlenecks (e.g., two tasks conflict) that prohibited meeting all guaranteed task deadlines. For a future CIRCA-II implementation, we also proposed a method by which a more general multi-resource scheduler can heuristically identify a bottleneck task to help guide the planner through a dynamic-backtracking [24] search for a schedulable plan. We expect this approach to increase the efficiency of plan development and minimize the value for $P_{removed}$, thereby minimizing the number of "removed" states.

Ignoring improbable states greatly facilitated schedulable plan development but jeopardized system safety when an unlikely state was actually encountered. In this dissertation, we developed a state-space classification that extended beyond the reachable state set to all states that were modelable from the symbolic set of state features and values. During planning, important "unhandled" states were enumerated and tests were built into each plan to ensure that they were identified as they occur during execution. We specifically detected three classes of unhandled states: removed states that were unlikely thus ignored during planning, imminent-failure states that led to failure but were not considered reachable unless a transition was modeled incorrectly or was missing from the knowledge base, and deadend states that were safe but did not lead to any goal. The CIRCA-II architecture included a plan cache to allow real-time plan retrieval and switching when required. The planner constructed a set of safety-critical contingency plans offline (i.e., prior to the system entering its dangerous environment) then retrieved and executed each as needed to react to unhandled states. We ultimately could not circumvent the limited-resource problem when the plan cache grew so large that contingency plans could not be retrieved in time to avoid failure. However, we were able to guarantee real-time failure avoidance even with a non-real-time planner for many situations in which a single control plan would fail.

We applied CIRCA-II to the challenging problem of achieving safe, fully-automated aircraft flight. We first described simulation results in which we attached CIRCA-II through a low-level control system to the ACM F-16 flight simulator [58] and fully-automated a complete flight. Although the domain knowledge only contained a token set of emergency situations, we illustrated the ability of CIRCA-II to react in real-time to critical but improbable in-flight situations such as a "gear-up-failure" by detecting the failure and retrieving a contingency plan that reacted to the situation appropriately and in real-time. We also presented results from a recent UCAV (Unmanned Combat Aerial Vehicle) demonstration that further illustrated CIRCA-II's ability to minimize the probability of failure, which in this case was represented by the aircraft being struck with an enemy missile. We described ongoing work to implement and test CIRCA-II on the University of Michigan's Uninhabited Aerial Vehicle, both to further validate CIRCA-II algorithms and to improve our representation of the critical features and values required to issue appropriate high-level commands for a fully-automated aircraft.

Contributions

In Chapter I, we described a specific set of research contributions made by this dissertation. In the following paragraphs, we revisit these contributions and refer to specific techniques presented in this dissertation that substantiate our claims.

Probabilistic Planning

The stochastic planning community has traditionally focused on the development of optimal policies as defined in Chapter I. We introduce a new perspective to this classical problem in that we are not that concerned with plan optimality or even state probability accuracy. Instead, we require the computation of accurate action timing constraints (deadlines) for our hard real-time execution environment and utilize state probabilities only to prioritize the reachable state-space. We have devised and built into CIRCA-II a probabilistic planning algorithm (Chapter IV) that requires a substantially smaller knowledge base than does an equivalent MDP planner and provides an approximate state probability distribution. We believe our method will be attractive to researchers studying complex real-time problem domains in which approximate state probabilities are sufficient, especially if these researchers have been frustrated with the complexity required even to specify the MDP for large state-spaces that include state history information.

Multi-layer Architectures

Our combination of distinct planning and plan execution layers provides a good fit for CIRCA-II within the multi-layer architectures community. We introduce a "real-time" bias to our system, as did the original CIRCA, defining our layers in terms of their ability to guarantee hard real-time response (plan-execution) versus best-effort (planning). The contribution of this dissertation to the architectures community lies in the ability of our algorithms to explicitly detect unhandled states during plan execution and dynamically react to this information (Chapter V). This dynamic reaction occurs internally to the plan-execution system when real-time response is required and externally (by requesting a new plan) when a best-effort response is acceptable. AI architectures are increasingly being applied to remote systems requiring hard real-time response to a variety of situations. In the majority of deployed systems, real-time response is verified only through exhaustive software testing, as opposed to the CIRCA-based approach of planning then scheduling to analytically verify meeting real-time constraints. We have made the CIRCA approach more attractive by introducing the capability to automatically respond in real-time to "unexpected" situations, a desirable capability for automated systems that cannot easily be reprogrammed.

Planner-Scheduler Negotiation

Instead of designing a "planner that schedules" or a "scheduler that plans", CIRCA and CIRCA-II use a state-space planner to select a set of actions for a plan and a traditional real-time scheduler to fit these actions onto available resources. We contribute to the field of "integrated planning-scheduling systems" by presenting methods to efficiently communicate quantities between scheduler and planner (Chapter VI). We use feedback from the scheduler as a new approach to guide planner backtracking when tradeoffs are required to schedule a plan. Specific examples of this feedback include suggesting a probability threshold for "ignoring" states and identifying a bottleneck task to be modified or removed. We also introduce an algorithm for Quality-of-Service (QoS) negotiation during task scheduling that continues to gain momentum as a viable method for making scheduling tradeoffs within the real-time community.

Aircraft Automation

Current flight management systems employ sophisticated control algorithms but have little higher-level software to reason about emergency situations. By considering the automation problem from the "human-pilot " perspective, an approach that lends itself to symbolic feature-value representations and high-level actions, we have imparted emergency response capability to simulated autonomous flight. For our research, CIRCA-II functions as the high-level "expert pilot" for which we have written the necessary low-level control laws to allow automated flight from takeoff through landing. This simulator is proving to be useful as a research testbed and has already been adopted by other research groups within the real-time systems laboratory at the University of Michigan and at Honeywell Technology Center. The flight control community will largely ignore this work in its present form. As we employ our techniques in future UAV experiments, we hope to gradually gain recognition by demonstrating automated fault (unhandled state) recovery techniques that complement ongoing system identification research in fault detection and isolation. This work may be particularly important when applied to dangerous situations such as airframe icing in which existing flight management systems may exacerbate the problem (e.g., by delaying pilot detection).

As with most research, delving deeply into specific topics serves to uncover more issues than are solved. This dissertation contributes techniques to researchers studying complex, real-time control problems from the high-level mission planning and real-time plan-execution perspective. We plan to continue this work into the next millennium and can only hope that future research will be as fun and intellectually-stimulating as has been this adventure into the interdisciplinary world of AI, real-time systems, and aircraft automation.

Future Research Directions

Several research areas we have addressed in this dissertation remain open for future study. We have presented algorithms for making tradeoffs during the development of real-time control plans, and we have given examples illustrating how the algorithms improve upon existing methods. Below, we describe ongoing efforts to quantitatively evaluate the accuracy and performance of the algorithms presented in this dissertation. We also describe future additions planned for the CIRCA-II architecture that will further enhance system plan generation and real-time execution capabilities. Then, we consider future UAV work and more general aspects of research that must be addressed before commercial aircraft flight can be both safe and autonomous.

CIRCA-II Evaluation

Definitively evaluating and quantitatively comparing the capabilities of numerous AI architectures is a challenging topic that spurs much discussion within the AI research community. We have argued that CIRCA and CIRCA-II are both more capable of operating in hard real-time environments than are many other architectures because the CIRCA approach proves that real-time plan execution deadlines will be met via explicit task scheduling. Additionally, we have demonstrated that CIRCA-II has the ability to fully-automate a flight simulator during normal flight and a specific set of emergency situations. However, we also acknowledge the need for a more formal evaluation.

Because CIRCA-II is composed of several distinct modules, the simplest way to attack the evaluation problem is to consider each component as a stand-alone system. We are currently in the process of evaluating the CIRCA-II planner probability model and outline our approach below. Next, we describe the challenges associated with evaluating the performance of our unhandled state detection and response mechanisms followed by a discussion of future plans to better assess CIRCA-II performance gains due to our planner-scheduler negotiation algorithms.

The CIRCA-II stochastic model is designed to compute approximate state probabilities by averaging history effects over all parent paths into each state. This model is a relatively recent addition to the CIRCA-II software, replacing an earlier version that is of limited use when dependent temporal transitions and cycles are present in the state-space. We have verified that our algorithms provide the expected results for simple examples. Although analysis is not yet complete, we are beginning to more formally assess the stochastic model via individual algorithm proofs of correctness (e.g., least-squares error minimization) as well as a comparison of CIRCA-II-generated state probabilities with stochastic simulation results. To perform the experimental evaluation, we generate a number of unique state-space structures from a set of generic non-domain-specific CIRCA-II knowledge bases. The state probabilities computed by CIRCA-II are then stored and compared to the results of a battery of stochastic simulations based on the same state transitions and probability rate functions. With this combined approach, we hope to develop a better understanding of the different state-space structures in which the CIRCA-II stochastic model is accurate as well as specific structures (such as that identified in Chapter IV) which are not well-characterized by the CIRCA-II weighted-average probabilities.

We also are working to better evaluate our approach to detecting and reacting to unplanned-for world states. Our initial evaluation strategy will include a parametric characterization of situations in which CIRCA-II will and will not perform well due to this approach. Although we have not yet determined the exact variables we will include during our tests, we expect to include knowledge base properties such as number of failure states and *ttfs* normalized over all states and state transitions, state-space characteristics such as the presence of cycles and/or dependent *tts*, and final plan properties such as the percentage of hard versus best-effort TAPs and resource utilization required for this plan.

Following our introspective analysis of CIRCA-II performance, we then hope to compare CIRCA-II with other architectures to better assess its relative capabilities. The challenge with this approach is in describing domain knowledge "equally" in the different architectures, then to execute the systems in a "fair" set of environments. Simply stated, an unplanned-for state in CIRCA-II may be planned-for in architecture X (or vice versa), depending on exactly how the knowledge was specified. Additionally, since few architectures guarantee hard real-time response, one might easily bias the tests such that the tradeoffs made by CIRCA-II are absolutely essential for success. Alternatively, one might bias the tests such that even a soft real-time response system will be adequate, in which case CIRCA-II will always "lose" in the comparisons. Although we feel such comparisons are challenging, we hope to borrow or manually craft a set of hard real-time problems then collaboratively run and compile results to more convincingly situate CIRCA-II as a viable approach for real-time automation problems.

We have provided tools for supplying the CIRCA-II planner with useful feedback from the scheduler to guide the backtracking process when a proposed plan cannot be scheduled. We have provided examples that illustrate the utility of this approach, but have not yet identified the parameters that can be used to specify circumstances under which our heuristic algorithms will be beneficial versus detrimental for our heuristic feedback approach. We envision planner-scheduler negotiation tests that exercise CIRCA-II in two directions: plan generation and plan execution efficiency.

We will assess plan generation efficiency as a function of whether the scheduler was able to provide useful feedback (e.g., whether the suggested $P_{removed}$ threshold sufficiently reduces task scheduling requirements or whether the identified bottleneck task can actually be safely modified during dynamic backtracking). As with the unhandled state tests, we suspect the best route to such tests is to develop a set of contrived examples in which states have a substantial range of probabilities (for $P_{removed}$ consideration) and planned actions vary along several dimensions (e.g., number of states in which each action is selected, TAP worst-case execution times, etc.).

During plan execution, we can simultaneously analyze the effectiveness of the scheduler's choice for $P_{removed}$ along with the accuracy of the stochastic planning model. A non-zero value for $P_{removed}$ indicates that CIRCA-II will be making only probabilistic safety guarantees. However, since our state probabilities are approximate and only represent the likelihood of ever visiting each particular state, it is not straightforward to translate state probabilities into an overall estimate of success.[67] We hope to use the same contrived domain with its "environment" linked to the stochastic simulation referenced above. Then, we can gather actual statistics regarding the frequency with which the system finds itself in a "removed" state, thereby assessing whether our notion of $P_{removed}$ given our approximate probability model is realistic.

Due to the breadth of the work undertaken for this dissertation, we have not yet completed the extensive set of tests proposed in this section. Although the time spent on developing appropriate example problems and constructing the support software for CIRCA-II tests may be substantial, we are convinced such analysis is crucial to demonstrate the utility of our algorithms to the research community. We also expect our evaluation will guide us toward improvements in many of the CIRCA-II algorithms.

[67] In Chapter VI, we gave a simple example where only one unlikely path was removed. In this case, we could estimate the probability of success based on the probability of reaching the first state along the removed path. However, more generally, multiple paths will have low-probability states removed, in which case we currently have no mechanism for computing overall success probability.

CIRCA-II Enhancement

Numerous opportunities remain for further improving the ability of CIRCA-II to develop and execute real-time control plans. We have identified research areas that can benefit from future enhancements in the context of existing CIRCA-II algorithms throughout this dissertation. Below, we revisit these plus additional topics that will better allow CIRCA-II to interact with its real-time environment efficiently and safely.

- *Real-time Plan Development*. We have incorporated a plan cache in CIRCA-II to minimize the need for dynamic planning. However, real-time planning constraints may be required when the cache becomes full (i.e., adding more plans would compromise real-time retrieval guarantees) before responses to all dangerous states are cached. CIRCA-II planning involves several algorithms, including planning, scheduling, and iteration between the two when a proposed plan cannot be scheduled. For real-time response, all of these processes must be bounded. Appendix D describes a possible approach for bounding planning time in CIRCA-II. This approach relies on the existing CIRCA-II stochastic state-space temporal model and combines ideas from the anytime and design-to-time literature.
- *Comprehensive Plan Development*. To be truly exhaustive, CIRCA-II backtracking processes must search through all possible state-action combinations, all possible sets of guarantee flags and deadlines for each action, and all value assigned to threshold $P_{removed}$. Except when directed by scheduler feedback, CIRCA-II backtracking is largely an *ad hoc* procedure that guarantees each action is tried once from each state, but not in all combinations. In addition to backtracking improvements, there are specific cyclic state-space structures that CIRCA-II cannot describe with sufficient accuracy to develop a valid plan. In Chapter IV, we identified a simple state-space with multiple dependent *ttfs* and multiple cycles that neither CIRCA nor CIRCA-II

handles adequately. Appendix E describes this example and outlines a preliminary solution approach that may solve this entire class of problems without adding a completely separate model-checking algorithm to the planner.

- *State-space Pruning*. For our research, we have relied on state probability as the sole measure of importance when the planner is forced to ignore states due to schedulability constraints. Other parameters may also play an important role in assessing areas of the state-space to prune when required. Perhaps the most intuitive parameter is to consider is action cost in terms of plan-execution resource requirements, a quantity that we have begun to consider during "bottleneck task" identification in Chapter VI. Also, since we cannot assume CIRCA-II will develop plans that are valid indefinitely, we may ultimately require at least an abstract notion of the minimum (global) time before the system may first visit each state. Such a time horizon is discussed in terms of imposing real-time bounds on planning in Appendix D, but is also an important parameter to consider when selecting states to ignore since exclusively far-term states may be handled in future plans. We expect CIRCA-II may develop higher-quality plans more efficiently using a combination of state probability, time horizon, and action computational cost when reasoning about regions of the state-space to prune.
- *Automatic Subgoal Development*. CIRCA-II requires that the user specify a list of subgoals given in its knowledge base. A method for dynamic backtracking to potential subgoal-splitting points is described in [4], but this procedure can only succeed when finding intermediate states with no exiting temporal transitions. Honeywell researchers are approaching this problem by implementing a distinct subgoal-generation module to the original CIRCA Automated Mission Planner module (see Chapter II). We hope their efforts will yield a solution that is also feasible for the CIRCA-II stochastic state-space planning model.

- *Planner-Scheduler Negotiation*. In this dissertation, we have addressed planner-scheduler negotiation from the perspective of "planner proposes plan; scheduler succeeds or feeds back $P_{removed}$ and/or a bottleneck task; planner backtracks and proposes a new plan when scheduling fails". In the final section of Chapter VI, we introduced a method for QoS negotiation within a general real-time systems framework. We plan to investigate methods for incorporating QoS negotiation protocols into the CIRCA-II scheduler so that we can *distribute* plan adaptation between planner and scheduler when tradeoffs to permit plan scheduling must be performed.
- *Machine Learning in CIRCA-II*. We foresee two near-term uses for learning-based algorithms in CIRCA-II: improvements for TAP test generation and the ability to "remember" contingency plans between missions. We have exclusively employed ID3 for minimizing TAP preconditions. As discussed in Chapter V, this introduces a substantial cost for building imminent-failure state detection TAPs, as do other existing decision tree algorithms because they require states with fully-instantiated feature values. We have not yet identified a better classification algorithm for TAP test construction but are continuing the search. We have introduced a plan cache into CIRCA-II that is always created from scratch for each mission. We adopt this approach because we require real-time contingency plan retrieval, a process slowed when searching through extraneous cached plans. CIRCA-II may see substantial savings in initial plan development overhead if the cache *remembers* plans it expects to require for multiple missions, although we have not yet identified the statistics that should be gathered by the dispatcher when making the decision to keep or discard plans after each run.
- *Multi-resource, Fault-tolerant Plan Execution*. In Chapter VI, we describe a method by which the CIRCA-II planner develops plans that execute on multiple resources and are able to handle specific computational system faults (e.g., single CPU failure).

However, the CIRCA-II Plan Execution Subsystem has always resided on a single processor. Recently, we have ported the uniprocessor plan execution software to the QNX operating system and migrated to a thread-based execution approach. In Appendix A, we discuss ongoing efforts to extend the CIRCA-II QNX-based software toward multi-resource plan execution. After this work is complete, we will next implement software to detect computational resource faults and respond in hard real-time by retrieving a new control plan (if necessary) then dynamically allocating the resources on which that plan will execute.

Safe, Fully-Automated Aircraft Flight

We have demonstrated that CIRCA-II can successfully automate simulated flights, including response to a specific set of carefully-engineered emergency situations. We have also constructed the hardware and designed the software architecture required to automate the University of Michigan UAV. We are still developing UAV low-level control and state estimation software that is necessary before CIRCA-II can even be employed. When this software is complete, we will then incorporate CIRCA-II to address challenging problems associated with the construction of real-time flight mission plans that guarantee safety in the context of the aircraft's dynamic capabilities, which necessarily evolve as system failures or environmental events (e.g., engine-failure or airframe-icing) occur. Although the results of this research are difficult to predict at this early stage, we hope the continued collaboration between control and symbolic planning researchers for the UAV will more generally lead to principled methods for reasoning about the controllability properties of complex nonlinear dynamical systems from the high-level "expert systems" perspective.

Numerous stochastic planning procedures, including MDP and CIRCA-II algorithms, have been demonstrated primarily on toy problems. In fact, for many complex problem domains, it is difficult to actually characterize the probability distributions of various events occurring because these events have not yet been considered in an approximate reasoning framework. For automated flight, meteorological phenomena are some of the most important environmental factors that require careful consideration both during initial flight planning and also when changes occur during flight. Stochastic models are already used to predict the occurrence and location of critical weather events (e.g., wind shear, thunderstorms, icing conditions) as is evident by the probabilistic predictions given by weather forecasters.

We hope to incorporate stochastic weather models into future CIRCA-II knowledge bases used for flight plan development. This research approach will yield two important results. First, it will allow us to better assess whether real-world data can be accurately incorporated into our state transition probability rate function format. Next, due to the dynamic nature of weather patterns, we will also be able to carefully study the effects of imprecise knowledge (i.e., outdated meteorological statistics) during subsequent plan execution, giving us a practical avenue for testing the CIRCA-II capabilities to detect and recover from the unhandled states resulting from weather pattern changes.

As we add complexity to our CIRCA-II knowledge base for fully-automating flight, we have observed that the likelihood of many important world events ranging from adverse weather to nearby air traffic may significantly vary even after the aircraft enters its environment. In CIRCA-II, we have assumed we could build plans to detect and react to each of these events given a static and general knowledge base. However, as our domain model complexity increases, we may eventually benefit from dynamic knowledge modification which would then require online construction of new plans. We have not yet begun to consider the practical implications of such a strategy except to note that CIRCA-II must then exhibit hard real-time planning response time and also should reason about minimally-modifying existing plans to maximize efficiency.

One important lesson we learned during this research is that safe, automated flight requires the solution of a multitude of research problems. We plan to continue our efforts to automate traditionally pilot-oriented tasks and forge collaborative ties with complementary research areas to ultimately achieve safe, fully-automated flight, even though the author of this dissertation may be very old when such automation is finally accepted.

APPENDICES

APPENDIX A

CIRCA-II C++ IMPLEMENTATION

This appendix is intended to serve as a reference for the CIRCA-II software and includes suggestions for upgrades in future work. Chapter III of this dissertation described the overall CIRCA-II components and their functions. However, we did not focus on any details of the software that weren't specifically designed to support the research presented in this thesis. Here, we present implementation details that will be critical as CIRCA-II undergoes further development, focusing on items that we already foresee a need to re-design or modify.

We organize this appendix into sections corresponding to modules within the CIRCA-II software. We first describe the CIRCA-II Planning Subsystem which will execute offline to develop the startup-set of nominal and contingency plans then online when required for reacting to unhandled states. We look at modules within the planner which would benefit from upgrades, including the algorithms used for action selection, backtracking, and building unhandled state detection tests. Next, we describe the implementation details of the CIRCA-II Plan-Execution Subsystem. As discussed briefly in Chapter VI, we will be migrating this software to a multi-resource environment and integrating the monitoring software necessary for detecting computational resource failures. The multi-resource CIRCA-II Plan-Execution Subsystem is slated to be implemented on the QNX real-time operating system, and we hope this conversion will more closely link CIRCA-II with recent research in dynamic resource allocation and scheduling from the real-time community.

Planning Subsystem

The CIRCA-II Planning Subsystem performs all the "unbounded" planning and scheduling tasks required to develop mission plans that will later execute in hard real-time on the CIRCA-II Plan-Execution Subsystem. Figure A-1 shows the algorithm used by the Planning Subsystem, including the basic plan development and plan-execution interface procedures. Note that the algorithm presented in this appendix is simply a more detailed version of the high-level Planning Subsystem described in Chapter III.

Upon startup, CIRCA-II builds the nominal and contingency plans it expects to require during plan execution. The initial (startup) state and list of task-level goals (subgoals) must be included in the domain knowledge base, along with the action and temporal state transitions. As shown in Figure A-1, nominal plans are developed from initial state(s) and a selected subgoal (Step 1) using a basic forward-chaining planning algorithm (Step 2), which employs best-first search based on the state probabilities computed from the algorithm described previously in Chapter IV. For each state, the planner selects actions (and their deadlines) required to preempt any temporal transitions to failure (*ttfs*) or a "best-effort" action (if required) to achieve task-level goals. When state expansion is completed, for each action, ID3 [24] is used (Step 3) to build a minimized "test" to determine whether that action should execute, with the set of states in which that action was selected as positive examples and all other "reachable" states as negative examples. Each minimized test and associated action is compiled into a TAP (test-action pair), with associated deadline (if any) set to the "worst-case" value for all states requiring that action.

```
1.  Set open list to initial state(s); select first subgoal from a pre-
     specified "mission goals" list.
2.  While (states above probability threshold $P_{removed}$ on open list)
        o  Select next "best" (highest probability) state
        o  Choose action (if any) based on failure avoidance and goal
           achievement
        o  Update all state probabilities (with Chapter IV algorithm)
        o  Move expanded state from open to closed list
   3.  Compile TAPs and hard real-time action deadlines
    in all expanded states
4.  Build set of "unhandled" state detection TAPs
5.  Schedule list of all "guaranteed" TAPs and maximize "if-time" TAP
     execution frequency
6.  If (No schedule possible for guaranteed TAPs)
        o  Resource scheduler suggests TAPs (based on utilization and
           priority) for removal
        o  Go to Step 2 for Backtracking, removing/relaxing period for
           time-consuming TAP(s) when possible, incrementing probability
           threshold $P_{removed}$ for "removing" (ignoring) states otherwise
7.  Download plan to Plan Cache with ID3-minimized "initial state test"
     as index
8.  If (time-critical "unhandled" states exist for this plan)
        o  Go to Step 2 with initial states (and open list) set to
           remaining time-critical "unhandled" state set; set goal to NIL
           (failure avoidance only)
9.  If (more subgoals)
        o  Select next subgoal from "mission goals" list
        o  Set initial state(s) to goal state(s) from last subgoal plan;
           set open list to initial state(s)
        o  Go to Step 2
10.  If (plan execution not started yet)
        o  Send "start" message to Plan-execution Subsystem
11.  Wait for message from Plan-execution Subsystem
12.  If (unhandled state message received)
        o  Set initial state(s)  to unhandled state and its "descendants".
           The set of descendants is  produced using  state expansion with
           the unhandled state as the initial state and the  currently-
           executing plan for all actions.
        o  If (one or more descendants is failure)
               - Set subgoal to NIL (failure avoidance only); Go to Step 2
        o  else
               - Select subgoal for replanning; Go to Step 2
13.  If (plan switch message received)
        o  Log new executing plan
        o  If (new plan is failure avoidance only)
               - Set initial state set to all states reachable from
                 executing plan
               - Select subgoal for replanning and Go to Step 2
14.  Go to Step 11
```

Figure A-1: CIRCA-II Planning Subsystem Algorithm.

The set of unhandled state detection TAPs is created (Step 4) using the algorithm from Chapter V, with deadlines set such that “removed” and “imminent failure” states will be detected before failure can occur. Finally (Step 5), the scheduler attempts to build a cyclic schedule for all guaranteed TAPs, with “if-time” (best-effort) TAPs filling any slack time. If scheduling fails (Step 6), the scheduler suggests removal of “bottleneck” TAP(s) and provides an estimate of the degree of failure (over-utilization), as described in Chapter VI. The planner backtracks (to Step 2) and continues (Steps 2-6) until a schedulable plan is produced.

When the plan is complete, it must be downloaded (Step 7) to the Plan-execution Subsystem to be stored in the cache. The most accurate index to this plan is the complete list of initial states. However, since this list could be very large and expensive to search through, and we require real-time cache access, we employ ID3 for building a minimal index to each plan, using the plan’s initial states as positive examples and all other states identified so far as negative examples.

Next, the planner builds contingency plans required for timely response to any “unhandled” state for which a time-critical response is required. This planning process begins (Step 8) with the set of initial states containing all of the imminent-failure states and the “dangerous” removed states which have been produced during development of this nominal plan (Step 4). Contingency planning assumes a “NIL” goal, indicating the plan need only include failure avoidance actions. In the best-case (i.e., few “dangerous” states), one schedulable contingency plan can handle all these states. Generally, though, on the first pass (Steps 2-8) only some of the time-critical unhandled states will be handled by the generated contingency plan. Contingency plan development continues iteratively (Steps 2-9) until schedulable plans exist for all time-critical unhandled states associated with the current nominal plan, or until real-time plan retrieval time limits on the plan cache prohibit further contingency plan storage.

Once the first nominal and associated contingency plans have been downloaded, further offline planning iterations are initiated with the remaining subgoals (Step 9) until nominal and contingency plans to achieve all subgoals and avoid failure in time-critical states have been produced. Next, the CIRCA-II Planning Subsystem notifies the Plan Dispatcher that it may begin executing the first plan (Step 10), at which time any remaining planning is dynamic (online) and prompted by messages received from the Dispatcher (Step 11), which include notification of an unhandled state (Step 12) and execution of a new plan (Step 13). An unhandled state message is received (Step 12) only if the Plan Cache has no nominal or contingency plan to react to this state, so the planner must produce a new plan that can respond appropriately. Because the world state (environment) can change while the planner is deliberating, the initial state(s) for this new plan must include both the unhandled state and its descendants. The possible descendant set is built using the planner's state expansion algorithm, with the fed-back unhandled state as the sole initial state, TAPs from the currently-executing plan as action transitions, and all temporal transitions from the knowledge base. If one of these descendants is failure, replanning occurs with the goal of failure-avoidance only to minimize planning time; otherwise, the unhandled state is a "deadend", so a new task-level goal is selected and full replanning occurs. Note that, when a dangerous state (e.g., a state matching a *ttf*) requires dynamic replanning, CIRCA-II can provide no real-time failure-avoidance guarantees. However, such a situation may occur in the worst case, so CIRCA-II attempts to respond and if "lucky" (i.e., planning and scheduling time are brief; *ttf* delay is longer than minimum) may "coincidentally" succeed. We address possibilities for implementing algorithms to bound planning time in Chapter IX of this dissertation.

Below, we discuss details of specific planning algorithms that we expect to require modification in future research. First, we describe the procedure by which the CIRCA-II planner selects the "best" action for each state and highlight methods that may be used to improve this procedure. Next, we describe algorithms available for backtracking during planning and recommend future improvements to both move the planner toward exhaustive backtracking as well as to better define the parameters over which the planner backtracks.

Action Selection

When a state is expanded, all actions with preconditions matching this state are added to a "candidate" list. Then, the planner calls an action scoring function which returns a numerical value associated with the input $\{s_k, ac_i\}$ pair. The original CIRCA architecture [53] uses a multi-level lookahead search to score each action, increasing the score for each downstream goal state and decreasing the score for each *ttf*. Several problems arise from this method. First, downstream states may not yet have actions planned for them, so there is no way to factor the likelihood of each goal state or *ttf* into this equation. In fact, either goal or failure states may be ultimately preempted by actions, resulting in an inaccurate initial score estimate. Additionally, if the *n*-level lookahead terminates even one step away from a goal state or *ttf* (i.e., at level $n+1$), the action scoring is unaffected.

Due to these difficulties, when we first adopted CIRCA-II for this research, we radically jumped to the conclusion that the *n*-level lookahead was not really giving significantly better results than would a 1-level lookahead, especially since the computational requirements for the *n*-level lookahead were nontrivial for large *n* and branching factor (i.e., number of *tts* matching each state). Therefore, in CIRCA-II we implemented a strictly one-level lookahead and measured action value based on immediate-descendant *ttfs* and goal states. Although we intended to revisit the action-scoring issue as part of this dissertation research, as of the writing of this dissertation all ideas remain in the highly conceptual phase, some of which we outline below.

In the above paragraphs, we paint the original CIRCA action scoring procedure in a negative light. However, a search-based procedure is really the only available option for a planner with a knowledge base containing only state transitions specified by symbolic-valued preconditions and postconditions. As an example, consider a feature "altitude" with values "zero", "low", "medium", and "high". Now, consider a knowledge base with an action "setup-climb" and temporal transitions to model the actual altitude changes. In order for a search-based scoring function to notice that the "setup-climb" action will lead to a goal state with "high" altitude from the current state with "zero" altitude, it must perform at least a 4-step lookahead. At the first step (level 0 to 1), a feature "climbing" is set to true, then at each subsequent level, the value of altitude progresses from "zero (level 1)->low (level2)->medium (level 3)->high (level 4)". Thus, if *n* for the *n*-level lookahead is set to 3 or less, the planner will not give the "setup-climb" any credit for moving the system "in the right direction".

For accurate action scoring, we really would like to have a measure of *proximity* for feature values relative to each other. For continuous numeric values such as altitude, this would simply allow the action scoring mechanism to note that a value of "low" or "medium" is closer to "high" than a value of "zero", thus it could add value for transitioning from "zero" to one of these values without explicitly performing the lookahead search. This functionality could be added to CIRCA-II by modeling the subset of features that represent continuous values with the median numeric value previously represented by a symbol. As a first step, these values could then be used to numerically judge feature distance from a goal or failure state, where distance could either be estimated as a feature value difference (the natural default) or else could be returned from a function that converted two feature values into a distance estimate. Then, either the 1- or n-level lookahead procedures might augment the distance-based estimate for inherently-symbolic feature values (e.g., "true" or "false").

We have not yet even developed concrete examples that argue for the combined distance/lookahead algorithm proposed here, thus we cannot yet assess whether this particular approach will find utility in CIRCA-II. However, we believe this avenue should at least be investigated in future work, because our past experiences with the existing action scoring algorithms in both CIRCA and CIRCA-II have indicated a definite need for future improvement.

Backtracking

The original CIRCA planner [53] expands states in depth-first order and performs *chronological backtracking* to the last action choice point whenever a *ttf* cannot be preempted, no goal state is reachable after state-space expansion terminates, or a plan cannot be scheduled. As backtracking progresses, each action with matching preconditions will eventually be selected for each state. However, this procedure is not exhaustive because the software does not try all possible *combinations* of actions for all expanded states.

If we were to implement the above procedure into the probabilistic CIRCA-II planner that performs its search in best-first order, chronological backtracking would not be so straightforward to implement because the planner jumps throughout the state-space rather than progressing down individual paths until they terminate or loop back to an expanded state.

Thus, we looked for alternate algorithms that might better enhance basic CIRCA-II capabilities. As described in [4], CIRCA-II currently utilizes a dynamic backtracking [24] algorithm based on traversing path-vectors that lead throughout the expanded state-space (and potentially to unexpanded child states of expanded parents). In this dissertation, the primary references to CIRCA-II backtracking occur when a plan cannot be scheduled. In Chapter VI, we describe an algorithm that identifies a "bottleneck" task.

Dynamic backtracking allows the planner to go directly to states for which this task's action was planned. This results (for the average case) in a much more efficient task modification/removal than would backtracking chronologically until that task is modified or removed in all states for which it must preempt a *ttf*.

The dynamic backtracking procedure is also not exhaustive, thus CIRCA-II may also fail to find a valid plan that could have been discovered with exhaustive backtracking. Additionally, individual action deadlines can significantly affect state probabilities, so different sets of states may be considered reachable (given preemption threshold P_{thresh} and removed-state threshold $P_{removed}$) using the same set of planned actions if the deadline for one or more of these actions is modified. We recommend that future CIRCA-II researchers implement the backtracking modifications required to account for these and other issues unique to development of real-time control plans.

Plan Execution Subsystem

As described in Chapter III, the Plan Execution Subsystem in CIRCA-II is responsible for reliably meeting all deadlines required for failure avoidance as dictated by the planner. Figure A-2 shows the basic top-level algorithm employed on the Plan Dispatcher module of the Plan Execution Subsystem (see Figure 3-1). Upon startup, no plan is executing, and CIRCA-II assumes the system will be indefinitely “safe” so long as no plan is begun (e.g., an aircraft sitting at a terminal gate prior to flight). The plan cache waits until the planner downloads all nominal and contingency plans (Step 1). As received, each plan is stored and indexed by plan type (nominal or contingency), as well the ID3-minimized test to determine whether a state matches the initial state set for that plan. Each contingency plan is also matched with a nominal plan to increase real-time plan retrieval efficiency.

Once all plans developed “offline” have been received, the planner will send a start message, at which time the first nominal plan received will begin execution (Step 2). After plan execution begins, any operation of the plan cache is explicitly controlled by received planner messages (Step 5) or executing TAPs that detect goal or unhandled states (Steps 3 and 4). When a TAP detects an unhandled state (Step 3), the plan cache searches its “contingency” plans for a match. If a match is found, this plan begins execution to “buy time” for the goal-oriented replanning required for the “unhandled state” deviation, and the cache sends a message notifying the planner of the plan switch. Otherwise, the state is not considered as time-critical as others for which contingency plans have been developed, so the old plan keeps executing while replanning occurs.

```
1.  While (no EXECUTE message received from Planner)
      °  As downloaded, add  plans to cache, indexed by decision tree
         and plan type (nominal/contingency) for state-to-plan matching
2.  Begin execution of first plan received as a Plan Executor process
3.  If (plan cache gets “unhandled state” message from Plan Executor)
      °  Search “contingency” subset of cache for matching plan
      °  If (plan found)
             - Begin execution of this plan on Plan Executor
             - Notify planner of contingency plan, since goal-
               achievement replanning may be required
      °  Else
             - Feedback unhandled state information to planner for
               replanning
4.  If (“goal achieved” message received from Plan Executor)
      °  Search cache for nominal plan to achieve next subgoal (indexed
         by current goal=next initial state)
      °  Place contingency plans matched with this nominal plan at top
         of queue for fast retrieval
      °  Begin execution of nominal plan on Plan Executor
      °  Notify planner of executing plan
5.  If ("add plan" message received from Planner)
      °  Add new plan to cache, indexed by decision tree and plan type
         for state-to-plan matching
      6.  If (EXECUTE message received from Planner)
      7.     -  Begin execution of this plan on Real-time Plan Executor
      °  Add new plan to cache, indexed by decision tree and plan type
         for matching Plan Executor feedback messages
7.  Go to Step 3
```

Figure A-2: CIRCA-II Plan Dispatcher Algorithm.

When a TAP detects a goal state (Step 4), the plan cache searches its "nominal" plans for a match with the goal state, since the appropriate next plan has an initial state equal to the old plan's goal state. Provided execution has proceeded as expected (i.e., the executing plan was also a nominal plan), a new plan will be found in the cache and will subsequently execute. Otherwise (i.e., a contingency plan is already executing), no goal state detection TAP will be included, so no goal state message will be produced. "New plan" messages received from the Planning Subsystem (Step 5) most often occur because an unhandled state has prompted replanning. In this case, the plan cache has been waiting for the new plan and it begins execution immediately. Alternatively, it is possible that the planner must develop a sequence of new plans during replanning (e.g., when one plan alone cannot redirect the system to the set of nominal goal-achievement plans), in which case the plan cache will store the plan as it did the original downloaded set (in Step 1).

The critical components for real-time operation of the Plan-execution Subsystem can be identified in terms of the Figure A-2 algorithm. Because Steps 1 and 2 occur during the startup period of "indefinite safety", they need not occur in hard real-time thus are of no concern from a timeliness perspective. Step 4 also need not occur in hard real-time because achieving task-level goals in CIRCA-II is strictly a best-effort endeavor, and retrieving a nominal plan is part of the goal-achievement process. Step 3, the search for a contingency plan and start of its execution, is the crucial part of the plan dispatcher that must execute in guaranteed real-time, as the entire purpose of the contingency plan set is to guarantee failure avoidance in improbable situations. Step 5 ordinarily does not require hard real-time guarantees, since it is most often associated with the receipt of plans that redirect the system to its task-level goals. However, CIRCA-II uses contingency plans to "buy time" for replanning (see Chapter V), and the amount of time it can "buy" may not be infinite. In such cases, one might consider Step 5 to benefit from hard real-time execution guarantees, although such guarantees will mean little since Step 5 relies on the “unbounded” planner for reactive plan development.

Below we look at two aspects of the CIRCA-II Plan Execution Subsystem that will most likely be studied in future work. First, we consider the implementation of the *If-time Server* used to execute best-effort (i.e., if-time) TAPs, then we consider future work to convert CIRCA-II Plan Execution to a multi-resource platform running the QNX real-time operating system.

***If-time Server* Implementation**

CIRCA-II executes best-effort (goal-achievement) TAPs only when extra time is available during execution of a guaranteed TAP schedule. As described in Chapter III, all best-effort TAPS execute under a "special" TAP named the *if-time server*. During the plan scheduling process, the scheduler attempts to maximize best-effort TAP execution by inserting the if-time server as frequently as possible into the guaranteed TAP schedule, setting the if-time server worst-case execution time (wcet) to the largest wcet of all best-effort TAPs. Regardless of whether the if-time server fits into this guaranteed schedule, the plan executor still executes the if-time server TAP whenever it finds a slack time interval (i.e., period of time between scheduled TAP completion and the wcet slot it was allotted in the schedule).

When invoked, the if-time server's job is to select a best-effort TAP to execute. This TAP must fit into the available slack time interval (equal to the if-time server wcet when the if-time server is called as part of the guaranteed TAP schedule). The if-time server would ideally also be "fair" regarding which TAP to execute next. In this section, we present three algorithms currently available for the if-time server in CIRCA-II.[68]

[68] The user must define a constant in the plan executor source code to select between the three available if-time server algorithms.

The original CIRCA used a simple *round-robin* strategy for selecting the next best-effort TAP to execute, as shown in Figure A-3. We have also implemented this algorithm in CIRCA-II. In the round-robin algorithm, the if-time server maintains a pointer to an element of the best-effort TAP list. When invoked, the if-time-server checks whether the current best-effort TAP (i.e., the TAP referenced by the pointer) will fit into the available slack time. If so, this TAP executes, the pointer moves on to the next TAP, and that next TAP is tested to check whether it fits into the available slack time. Whenever the remaining slack time is less than the wcet of the next TAP to be executed, control returns to the calling program.

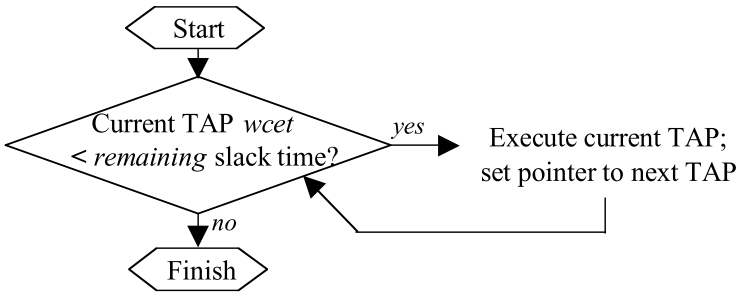

Start
Current TAP *wcet* < *remaining* slack time?
yes
Execute current TAP; set pointer to next TAP
no
Finish

Figure A-3: Round-Robin *If-time Server*.

Figure A-4 illustrates a slight modification of the round-robin protocol we have labeled the *modified-round-robin* if-time server algorithm. This algorithm is also available in CIRCA-II (as well as in certain versions of the original CIRCA). In this algorithm, a pointer is also maintained to the next item on an if-time server list. When invoked, the if-time-server again tests whether the current TAP will fit into the available server time slot. If so, it is executed and the pointer is incremented as with the round-robin algorithm. However, if not, instead of "giving up", the modified-round-robin protocol increments the pointer and checks whether the next TAP on the list will fit into the available time slot. As shown in the figure, this procedure continues until either the remaining available time falls below a minimum value or else the pointer cycles back to where it began during this if-time server function call. This algorithm is viewed as an improvement in that more best-effort TAPs will execute. However, the modified-round-robin server never resets its TAP pointer, so the TAP originally referenced by the pointer may not have executed during this invocation and will not be immediately referenced during the subsequent if-time server call. As a result, best-effort TAPs with relatively large wcets may not execute as frequently as would they would within the traditional round-robin if-time server.

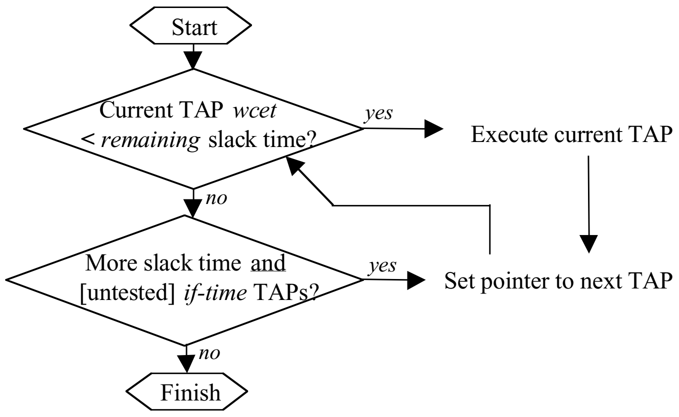

Start
Current TAP *wcet* < *remaining* slack time?
yes
Execute current TAP
no
More slack time and [untested] *if-time* TAPs?
yes
Set pointer to next TAP
no
Finish

Figure A-4: Modified-Round-Robin *If-time Server*.

Figure A-5 depicts an if-time queue algorithm, the newest of the if-time server algorithms and available only in CIRCA-II (both the UNIX and QNX versions). In this algorithm, the if-time server maintains a queue. Upon invocation, the server checks whether the TAP at the top of the queue will fit into the slack interval. If so, this TAP executes and migrates to the end of the queue. Regardless of whether the TAP at the top of the queue executes, the pointer is set to the next TAP in the queue which then executes if time is available. As with the modified-round-robin algorithm, this process continues until either the slack time expires or the pointer moves sufficiently deep into the queue to reach a TAP that has already been executed.

We do not yet have analytical results to compare these three algorithms, but intuitively find the if-time queue algorithm the best approach because it treats the best-effort TAPs fairly, at least in terms of attempting to execute all best-effort TAPs at as close to the same frequency as is feasible given the slack interval size and TAP wcets.

Work by others has begun to analyze whether the if-time server can use quantities such as TAP priority or probability from the planner and TAP last-finished-execution-time (lfet) to better order server selection of TAPs to execute. However, no definitive algorithms or analytical comparisons of such methods exist to-date. As CIRCA-II continues its migration to the QNX real-time operating system, the if-time server may disappear in favor of real-time dynamic [priority] scheduling algorithms. If/when this change occurs, these quantities can be incorporated into a cost function that can be used as a priority input for the dynamic scheduler.

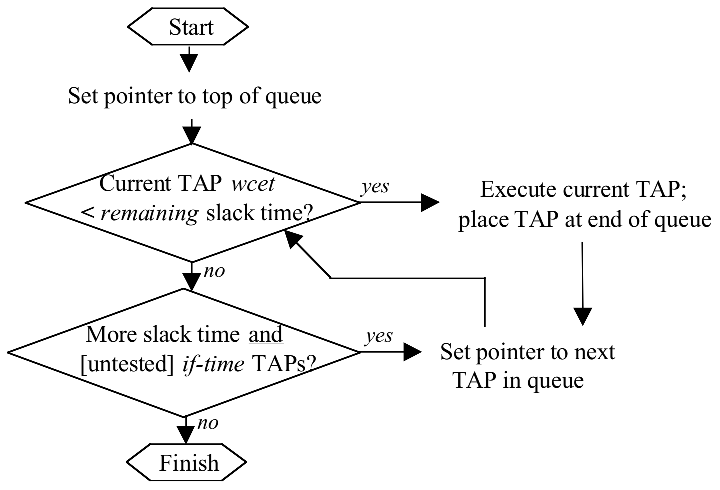

Start
Set pointer to top of queue
Current TAP *wcet* < *remaining* slack time?
yes
Execute current TAP; place TAP at end of queue
no
More slack time and [untested] *if-time* TAPs?
yes
Set pointer to next TAP in queue
no
Finish

Figure A-5: If-time Queue *If-time Server*.

O/S- and Domain-Dependent Software

To-date, the majority of CIRCA and CIRCA-II testing has been done on a UNIX-based platform (except for the QoS Negotiation work from Chapter VI). Unfortunately, realistic worst-case execution times are nearly impossible to predict in typical UNIX environments, so we have had little success in actually plotting performance results during plan execution. We have outfitted CIRCA-II so that it will execute under the QNX real-time operating system for the University of Michigan UAV project. As discussed in Chapter VII, the UAV requires that the complete CIRCA-II Plan-Execution Subsystem execute on one processor, thus the current QNX implementation, like its UNIX predecessor, still presumes a single-processor execution platform.

The UAV software is designed as a group of distinct processes spawned after startup. After CIRCA-II develops and caches its initial set of plans, the dispatcher spawns a Plan-Executor process for the first plan. As described with more detail in [65], a plan executor process begins by spawning separate processes for each TAP in its plan.[69] The main executor process then attaches a proxy to each TAP that is "kicked" each time that TAP should execute. Each TAP process sits idly waiting to be kicked and then executes. This design is similar to a multi-thread execution model often adopted by the real-time community. In future work, we hope to extend CIRCA-II to a multi-resource execution platform, as discussed briefly in Chapter VI. We believe the thread-based model currently implemented under QNX will facilitate the augmentation of the CIRCA-II Plan-Execution Subsystem to a multi-resource platform, although a substantial number of issues still need to be addressed, including the design and implementation of an algorithm to distribute dispatcher execution among the available resources and an algorithm to monitor the system for faults and reallocate TAPs to use the remaining resources.

---

[69] We presume feature values can be communicated easily to all processes. For our UAV, we have shared memory available to all executing processes. This memory contains all feature value data (determined during state estimation) so that determining feature values consumes a trivial amount of time during TAP test execution. This situation may not be true in other domains in which feature tests may require the system to interact directly with its environment.

APPENDIX B
AUTOMATION OF THE ACM SIMULATOR WITH CIRCA-II

The algorithms presented in this dissertation were first tested for fully-automated "flight-around-the-pattern" (see Chapter VII) during which we introduced anomalous events and costly actions to challenge the ability of CIRCA-II to develop a schedulable plan. In this appendix, we first include one version of a knowledge base capable of guiding the aircraft around the pattern, with each pattern "leg" specified as a separate subgoal for which a plan is developed. The initial ACM tests performed using this particular knowledge base relied on an earlier version of a CIRCA-II probabilistic model that is described [8]. We did not re-run these experiments because their main purpose was to illustrate procedures to detect and react to unplanned-for states, and state probabilities were used simply to identify unlikely states to "remove".

Following the knowledge base, we include the nominal plans produced that when executed in sequential order guide the aircraft around the pattern to a safe landing. In the specific scenario presented in this appendix, the reaction to "gear-up-on-final-approach" could not be scheduled into the final approach plan, thus the nominal final approach plan contains a TAP to detect a "gear up" event as a removed state. CIRCA-II develops and caches a contingency plan to handle the "gear up" emergency should it arise, then the dispatcher retrieves and executes this plan if it is required during flight.

Knowledge Base

```
#######################################################################
#
#  CIRCA_C Knowledge base file
#
#  Written:  Ella M. Atkins, April 1997
#  Last Modified:  April 1997
#
#  Use this format to create CIRCA (C++ version) knowledge
#  base files.  The program "make_kbase" will use this data to
#  create a more efficient program for CIRCA to use during planning.
#
#  Note:  "make_kbase" is made easier by order-dependent data
specification.
#    1.  Initial State(s),
#    2.  Subgoal(s),
#    3.  Feature wcet's.
#    4.  Action transition(s),
#    5.  Temporal transition(s).
#    6.  Temporal transition probability function definitions.
#
#  **** All names (features, values, transitions) must be <=20 chars
each,
#       with no "special" embedded characters (e.g.,space,.,',",+,-
,*,/).
#       Feature names and values must begin with a capital letter;
#       feature names and action transition names must correspond to
#       function names for plan execution (in domain.cpp).
#
########################################################################
#

########################################################################
#
#
# Section 1:  Initial States
#
# In the following lines, specify all features for each initial state,
# marking beginning with "begin initial_state:" and ending with "end"
# for each different possible initial state.
#
#
begin initial_state:
  Failure = False;  Traffic = False; Tornado = False; Hurricane = False;
  Swerve = False; Avoiding_Traffic = False; On_Course = True; Gear =
Down;
  Altitude = Zero; Heading = S; Location = Fix0; Obs = Fix0; Nav_Freq =
Land;
end

# (Additional initial states may be specified here using
#  this same "begin-end" format)
```

```
########################################################################
#
#
#  Section 2:  Subgoal(s)
#
#  Specify all subgoal features (need not be a complete list) and
#  preconditions for achieving each subgoal, using the "begin-end"
format
#  shown below.  Note:  For now, the order of subgoals specified here
will
#  be the default order CIRCA uses during planning iterations.
#
begin subgoal:
  features:      Obs = Fix1; Location = Fix1; Heading = S;
  preconditions:  Location = Fix0;
end

begin subgoal:
  features:       Obs = Fix2; Location = Fix2; Heading = E;
  preconditions:  Location = Fix1;
end

begin subgoal:
  features:       Obs = Fix3; Location = Fix3; Heading = N;
  preconditions:  Location = Fix2;
end

begin subgoal:
  features:       Obs = Fix4; Location = Fix4; Heading = W;
  preconditions:  Location = Fix3;
end

begin subgoal:
  features:       Nav_Freq = Land; Obs = Fix0; Location = Fix0; Heading
= S;
  preconditions:  Location = Fix4;
end


########################################################################
#
#
# Section 3:  Feature WCET's
#
# In the following lines, specify all feature test
# worst-case execution times (wcets).
#
#
begin feature_wcets:
  Failure = 0.0;
  Traffic = 1000.0;
  Tornado = 1000.0;
  Hurricane = 1000.0;
  Swerve = 1000.0;
  Avoiding_Traffic = 1000.0;
  On_Course = 1000.0;
  Gear = 1000.0;
  Altitude = 1000.0;
  Heading = 1000.0;
  Location = 1000.0;
  Obs = 1000.0;
  Nav_Freq = 1000.0;
end


########################################################################
#
#
#  Section 4:  Action Transitions
#
#  Specify action transitions here, using similar "begin-end" format
#  as shown by example below.  Each must have a name, preconds,
postconds,
#  and wcet.
#
#  --  For preconds, specify a C-formatted test sequence enclosed by ()
like
#      that used as a test for "if ()".  This test sequence should
return
#      True if transition preconditions are matched.  Feature value
#      comparisons are given by:  (f[feature_name] == (or !=)
feature_value).
#  --  For postconds, specify C-formatted statements that change values
#      in an array f[] to their "new" postcondition values.
#
begin action:
  name:       climb_to_altitude
  preconds:   ((f[Altitude] == Zero) && (f[Nav_Freq] == Fly))
  postconds:  f[Altitude] = Pos;
  wcet:       5000
end

begin action:
  name:      turn_left_to_E
  preconds:  ((f[Heading] == S) && (f[Altitude] == Pos)
              && (f[Traffic] == False) && (f[Swerve] == False)
            && (f[Obs] == Fix2))
  postconds: f[Heading] = E;
  wcet:       5000
end

begin action:
  name:      turn_left_to_N
  preconds:  ((f[Heading] == E) && (f[Altitude] == Pos)
              && (f[Traffic] == False) && (f[Swerve] == False)
            && (f[Obs] == Fix3))
  postconds: f[Heading] = N;
  wcet:       5000
end

begin action:
  name:      turn_left_to_W
  preconds:  ((f[Heading] == N) && (f[Altitude] == Pos)
              && (f[Traffic] == False) && (f[Swerve] == False)
            && (f[Obs] == Fix4))
  postconds: f[Heading] = W;
  wcet:       5000
end

begin action:
  name:      turn_left_to_S
  preconds:  ((f[Heading] == W) && (f[Altitude] == Pos)
              && (f[Traffic] == False) && (f[Swerve] == False)
            && (f[Location] == Fix4))
  postconds: f[Heading] = S;
  wcet:       5000
end

begin action:
  name:      turn_left_to_Sb
  preconds:  ((f[Heading] == W) && (f[Altitude] == Pos)
              && (f[Traffic] == False)  && (f[Swerve] == False)
              && (f[Obs] == Fix4) && (f[Location] == Fix6))
  postconds: f[Heading] = S;
  wcet:       5000
end

begin action:
  name:      obs_set_fix1
  preconds:  ((f[Obs] == Fix0) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix0) && (f[On_Course] == True))
  postconds: f[Obs] = Fix1;
  wcet:       5000
end

begin action:
  name:      obs_set_fix2
  preconds:  ((f[Obs] == Fix1) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix1) && (f[On_Course] == True))
  postconds: f[Obs] = Fix2;
  wcet:       5000
end

begin action:
  name:      obs_set_fix3
  preconds:  ((f[Obs] == Fix2) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix2) && (f[On_Course] == True))
  postconds: f[Obs] = Fix3;
  wcet:       5000
end
```

```
begin action:
  name:      obs_set_fix4
  preconds:  ((f[Obs] == Fix3) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix3) && (f[On_Course] == True))
  postconds: f[Obs] = Fix4;
  wcet:       5000
end

begin action:
  name:      obs_set_fix4b
  preconds:  ((f[Obs] == Fix6) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix6) && (f[On_Course] == True))
  postconds: f[Obs] = Fix4;
  wcet:       5000
end

begin action:
  name:      obs_set_fix5
  preconds:  ((f[Obs] == Fix3) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix3) && (f[On_Course] == True))
  postconds: f[Obs] = Fix5;
  wcet:       5000
end

begin action:
  name:      obs_set_fix6
  preconds:  ((f[Obs] == Fix5) && (f[Nav_Freq] == Fly) &&
              (f[Location] == Fix5) && (f[On_Course] == True))
  postconds: f[Obs] = Fix6;
  wcet:       5000
end

begin action:
  name:      obs_set_fix0
  preconds:  ((f[Obs] == Fix4) && (f[Heading] == S) &&
              (f[Location] == Fix4) && (f[On_Course] == True))
  postconds: f[Obs] = Fix0;
  wcet:       5000
end

begin action:
  name:      Nav_Freq_fly
  preconds:  ((f[Nav_Freq] == Land) && (f[Altitude] == Zero))
  postconds: f[Nav_Freq] = Fly;
  wcet:       5000
end

begin action:
  name:      Nav_Freq_land
  preconds:  ((f[Nav_Freq] == Fly) && (f[Location] == Fix4) &&
              (f[Heading] == S) && (f[Obs] == Fix0) && (f[Altitude] ==
Pos))
  postconds: f[Nav_Freq] = Land;
  wcet:       5000
```

```
end

begin action:
  name:      avoid_traffic
  preconds:  ((f[Traffic] == True) && (f[Avoiding_Traffic] == False) &&
              (f[On_Course] == True) && (f[Swerve] == False))
  postconds:  f[Avoiding_Traffic]=True; f[Swerve]=True;
f[On_Course]=False;
  wcet:       5000
end

begin action:
  name:      course_correct
  preconds:  ((f[Swerve] == True) && (f[On_Course] == False) &&
            (f[Traffic] == False) && (f[Avoiding_Traffic] == True))
  postconds: f[Swerve]=False; f[On_Course]=True;
f[Avoiding_Traffic]=False;
  wcet:       5000
end


########################################################################
#
#
#  Section 5:  Temporal Transitions
#
#  Specify temporal transitions.  Use very similar format to that
described
#  for action transitions above, and illustrated by example below.
#  Each temporal transition must have a name, preconds, postconds, and
#  a prob_func (where each prob_func referenced must be defined in the
#  next section below).
#

begin temporal:
  name:      zero_altitude
  preconds:  ((f[Altitude] == Pos) && (f[Nav_Freq] == Fly))
  postconds: f[Altitude] = Zero;
  prob_func: z_altitude
end

begin temporal:
  name:      fly_fix0_to_fix1
  preconds:  ((f[Location] == Fix0) && (f[Heading] == S) && (f[Obs] ==
Fix1)
              && (f[Nav_Freq] == Fly) && (f[Altitude] == Pos) &&
              (f[On_Course] == True))
  postconds: f[Location] = Fix1;
  prob_func: fly_fixes
end

begin temporal:
  name:      fly_fix1_to_fix2
  preconds:  ((f[Location] == Fix1) && (f[Heading] == E) && (f[Obs] ==
Fix2)
              && (f[Nav_Freq] == Fly) && (f[Altitude] == Pos) &&
              (f[On_Course] == True))
  postconds: f[Location] = Fix2;
  prob_func: fly_fixes
end

begin temporal:
  name:      fly_fix2_to_fix3
  preconds:  ((f[Location] == Fix2) && (f[Heading] == N) && (f[Obs] ==
Fix3)
              && (f[Nav_Freq] == Fly) && (f[Altitude] == Pos) &&
              (f[On_Course] == True))
  postconds: f[Location] = Fix3;
  prob_func: fly_fixes
end

begin temporal:
  name:      fly_fix3_to_fix4
  preconds:  ((f[Location] == Fix3) && (f[Heading] == W) && (f[Obs] ==
Fix4)
              && (f[Nav_Freq] == Fly) && (f[Altitude] == Pos) &&
              (f[On_Course] == True))
  postconds: f[Location] = Fix4;
  prob_func: fly_fixes
end

begin temporal:
  name:      fly_fix4_to_fix0
  preconds:  ((f[Location] == Fix4) && (f[Heading] == S) && (f[Obs] ==
Fix0)
              && (f[Nav_Freq] == Land) && (f[Gear] == Down) &&
              (f[On_Course] == True))
  postconds: f[Location] = Fix0; f[Altitude] = Zero;
  prob_func: fly_fixes
end

begin temporal:
  name:      any_traffic_tt
  preconds:  ((f[Traffic] == False) && (f[Nav_Freq] == Fly)
              && (f[Swerve] == False) && (f[On_Course] == True)
              && (f[Avoiding_Traffic] == False))
  postconds: f[Traffic] = True;
  prob_func: any_traffic
end

begin temporal:
  name:      traffic_passes_tt
  preconds:  ((f[Traffic] == True) && (f[Swerve] == True) &&
              (f[Avoiding_Traffic] == True) && (f[On_Course] == False))
  postconds: f[Traffic] = False;
  prob_func: traffic_passes
end
```

```
begin temporal:
  name:      intercept_course_tt
  preconds:  ((f[Swerve] == False) && (f[On_Course] == False) &&
            (f[Traffic] == False) && (f[Avoiding_Traffic] == True))
  postconds: f[Avoiding_Traffic] = False;
  prob_func: intercept_course
end

begin temporal
  name:      gear_up_failure
  preconds:  ((f[Location] == Fix4) && (f[Nav_Freq] == Land) &&
            (f[Gear] == Down))
  postconds:  f[Gear] = Up;
  prob_func:  gear_up_failure
end


#  Temporal Transitions to Failure (TTFs)

begin temporal:
  name:      drive_into_ground_0
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix0)
              && (f[Obs] == Fix1))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      drive_into_ground_1
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix1))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      drive_into_ground_2
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix2))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      drive_into_ground_3
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix3))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      drive_into_ground_4
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix4)
              && (f[Nav_Freq] == Fly))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      drive_into_ground_5
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix5))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      drive_into_ground_6
  preconds:  ((f[Altitude] == Zero) && (f[Location] == Fix6))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      land_gear_up
  preconds:  ((f[Altitude] == Zero) && (f[Gear] == Up))
  postconds: f[Failure] = True;
  prob_func: drive_into_ground
end

begin temporal:
  name:      mid_air_collision_tt
  preconds:  ((f[Traffic] == True) && (f[Avoiding_Traffic] == False))
  postconds: f[Failure] = True;
  prob_func: mid_air_collision
end

begin temporal:
  name:      fly_way_off_course_1
  preconds:  ((f[Avoiding_Traffic] == True) && (f[Swerve] == True) &&
              (f[Traffic] == False))
  postconds: f[Failure] = True;
  prob_func: hopelessly_off_course
end

begin temporal:
  name:      fly_way_off_course_2
  preconds:  ((f[Traffic] == False) && (f[Swerve] == False)
              && (f[On_Course] == False) && (f[Avoiding_Traffic] ==
False))
  postconds: f[Failure] = True;
  prob_func: hopelessly_off_course
end

########################################################################
#
#
#  Section 6:  Probability function definition
#              (for temporal transitions)
#
```

```
#  Each of these functions should be written as a valid C++ function or
macro.
#  See example for format --> Assumes arguments (float time, float
prob);  if
#  prob set to <= 0.0, use time to compute prob; otherwise, use prob to
#  compute time.  Returns a float (either time or prob).
#
#  For returning impossible (beyond asymptote) probabilities, use
#  value FLT_MAX from <float.h>.
#

begin prob_func:

// z_altitude prob_func
float z_altitude(float time, float prob)
{
  if (prob <= 0.0) {
      if (time < 10000.0)  return(((float) time)/100000.0);
      else                return(0.1);
  } else {
      if (prob <= 0.1)   return(prob * 100000.0);
      else                return(FLT_MAX);
  }
}

// fly_fixes prob_func
float fly_fixes(float time, float prob)
{
  if (prob <= 0.0) {
      if (time < 20000.0)       return(0.0);
      else if (time < 200000.0) return(((float) (time - 20000)) /
200000.0);
      else                     return(0.9);
  } else {
      if (prob == 0.0)         return(0.0);
      else if (prob <= 0.1)    return(20000.0 + (prob * 200000.0));
      else                     return(FLT_MAX);
  }
}

// any_traffic prob_func
float any_traffic(float time, float prob)
{
  if (prob <= 0.0) {
      if (time < 10000.0)  return(((float) time)/100000.0);
      else                return(0.1);
  } else {
      if (prob <= 0.1)   return(prob * 100000.0);
      else                return(FLT_MAX);
  }
}

// traffic_passes prob_func
float traffic_passes(float time, float prob)
```

```
{
  if (prob <= 0.0) {
        if (time < 5000.0)        return(0.0);
      else if (time <= 50000.0) return(((float) (time - 5000))/50000.0);
      else                      return(0.9);
  } else {
      if (prob == 0.0)          return(0.0);
      else if (prob <= 0.9)     return(5000.0 + (prob * 50000.0));
      else                 return(FLT_MAX);
  }
}

// intercept_course prob_func
float intercept_course(float time, float prob)
{
  if (prob <= 0.0) {
        if (time < 5000.0)        return(0.0);
      else if (time <= 50000.0) return(((float) (time - 5000))/50000.0);
      else                      return(0.9);
  } else {
      if (prob == 0.0)          return(0.0);
      else if (prob <= 0.9)     return(5000.0 + (prob * 50000.0));
      else                 return(FLT_MAX);
  }
}

// drive_into_ground prob_func
float drive_into_ground(float time, float prob)
{
  if (prob <= 0.0) {
      if (time < 100000.0)  return(0.0);
      else                return(1.0);
  } else {
      return(100000.0);
  }
}

// mid_air_collision prob_func
float mid_air_collision(float time, float prob)
{
  if (prob <= 0.0) {
      if (time < 100000.0)   return(0.0);
      else                return(1.0);
  } else {
      return(100000.0);
  }
}

// hopelessly_off_course prob_func
float hopelessly_off_course(float time, float prob)
{
  if (prob <= 0.0) {
      if (time < 200000.0)   return(0.0);
      else                return(1.0);
  } else {
      return(200000.0);
  }
}

end
```

Plan Sequence Generated for *Flight-Around-the-Pattern*

Plan0: Takeoff and Fly Upwind Pattern Leg to FIX1

```
#################################
# plan0.txt
#
# Automatically generated by CIRCA planner.
#
#################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 4;
  NIF_TIME_TAPS=3;
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 9000.000000;
  if_time_tap0 = 3000.000000;
  if_time_tap1 = 9000.000000;
  if_time_tap2 = 6000.000000;
  tap1 = 12000.000000;
  tap2 = 12000.000000;
  tap3 = 12000.000000;
end

# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap2, tap0, tap3, tap0, tap1, tap0 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end
```

```
begin tap:
  name: tap1
  preconds:  (( ((Nav_Freq() == Fly) && (Swerve() == False) &&
(Traffic() == False) && (Obs() == Fix0) && (Altitude() == Zero)) ||
 ((Obs() == Fix1) && (Altitude() == Zero))))
  action:  climb_to_altitude();
end

begin tap:
  name: tap2
  preconds:  (( ((Obs() == Fix0) && (Altitude() == Zero) && (Swerve() ==
False) && (Traffic() == True)) ||
 ((Altitude() == Pos) && (Swerve() == False) && (Traffic() == True))))
  action:  avoid_traffic();
end

begin tap:
  name: tap3
  preconds:  (( ((Obs() == Fix0) && (Altitude() == Zero) && (Traffic()
== False) && (Swerve() == True)) ||
 ((Altitude() == Pos) && (Traffic() == False) && (Swerve() == True))))
  action:  course_correct();
end

begin tap:
  name: if_time_tap0
  preconds:  ((Heading() == S) && (Location() == Fix1) && (Obs() ==
Fix1))
  action:  run_new_plan();
end

begin tap:
  name: if_time_tap1
  preconds:  (( ((Swerve() == False) && (Traffic() == False) &&
(Altitude() == Pos) && (Obs() == Fix0))))
  action:  obs_set_fix1();
end

begin tap:
  name: if_time_tap2
  preconds:  (( ((Nav_Freq() == Land))))
  action:  Nav_Freq_fly();
end
```

Plan1: Turn and Fly Crosswind Pattern Leg to FIX2

```
##################################
# plan1.txt
#
# Automatically generated by CIRCA planner.
```

```
#
#################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 4;
  NIF_TIME_TAPS=3;
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 10000.000000;
  if_time_tap0 = 6000.000000;
  if_time_tap1 = 10000.000000;
  if_time_tap2 = 9000.000000;
  tap1 = 10000.000000;
  tap2 = 8000.000000;
  tap3 = 8000.000000;
end

# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap2, tap0, tap3, tap0, tap1, tap0 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (( ((Swerve() == False) && (Altitude() == Zero)) ||
 ((Obs() == Fix1) && (Swerve() == True) && (Altitude() == Zero))))
  action:  climb_to_altitude();
end

begin tap:
  name: tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == True))))
  action:  avoid_traffic();
end

begin tap:
  name: tap3
  preconds:  (( ((Altitude() == Pos) && (Swerve() == True) && (Traffic()
== False))))
  action:  course_correct();
```

```
end

begin tap:
  name: if_time_tap0
  preconds:  ((Heading() == E) && (Location() == Fix2) && (Obs() ==
Fix2))
  action:  run_new_plan();
end

begin tap:
  name: if_time_tap1
  preconds:  (( ((Obs() == Fix2) && (Heading() == S) && (Altitude() ==
Pos) && (Swerve() == False) && (Traffic() == False))))
  action:  turn_left_to_E();
end

begin tap:
  name: if_time_tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == False) && (Obs() == Fix1))))
  action:  obs_set_fix2();
end
```

Plan2: Turn and Fly Downwind Pattern Leg to FIX3

```
#################################
# plan2.txt
#
# Automatically generated by CIRCA planner.
#
#################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 4;
  NIF_TIME_TAPS=3;
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 10000.000000;
  if_time_tap0 = 9000.000000;
  if_time_tap1 = 10000.000000;
  if_time_tap2 = 9000.000000;
  tap1 = 10000.000000;
  tap2 = 8000.000000;
  tap3 = 8000.000000;
end

# Part 3:  Define schedule (by guaranteed TAP name).
```

```
begin tap_schedule:
  { tap2, tap0, tap3, tap0, tap1, tap0 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (!( ((Location() == Fix2) && (Heading() == N) && (Swerve()
== True) && (Altitude() == Zero)) ||
 ((Altitude() == Pos))))
  action:  climb_to_altitude();
end

begin tap:
  name: tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == True))))
  action:  avoid_traffic();
end

begin tap:
  name: tap3
  preconds:  (( ((Altitude() == Pos) && (Swerve() == True) && (Traffic()
== False))))
  action:  course_correct();
end

begin tap:
  name: if_time_tap0
  preconds:  ((Heading() == N) && (Location() == Fix3) && (Obs() ==
Fix3))
  action:  run_new_plan();
end

begin tap:
  name: if_time_tap1
  preconds:  (( ((Obs() == Fix3) && (Heading() == E) && (Altitude() ==
Pos) && (Swerve() == False) && (Traffic() == False))))
  action:  turn_left_to_N();
end

begin tap:
  name: if_time_tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == False) && (Obs() == Fix2))))
  action:  obs_set_fix3();
end
```

Plan3: Turn and Fly Base Pattern Leg to FIX4

```
##################################
# plan3.txt
#
# Automatically generated by CIRCA planner.
#
##################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 4;
  NIF_TIME_TAPS=3;
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 12000.000000;
  if_time_tap0 = 12000.000000;
  if_time_tap1 = 10000.000000;
  if_time_tap2 = 9000.000000;
  tap1 = 10000.000000;
  tap2 = 8000.000000;
  tap3 = 8000.000000;
end

# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap2, tap0, tap3, tap0, tap1, tap0 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (( ((Swerve() == False) && (Altitude() == Zero)) ||
 ((Obs() == Fix3) && (Swerve() == True) && (Altitude() == Zero))))
  action:  climb_to_altitude();
end

begin tap:
  name: tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == True))))
  action:  avoid_traffic();
```

```
end

begin tap:
  name: tap3
  preconds:  (( ((Altitude() == Pos) && (Swerve() == True) && (Traffic()
== False))))
  action:  course_correct();
end

begin tap:
  name: if_time_tap0
  preconds:  ((Heading() == W) && (Location() == Fix4) && (Obs() ==
Fix4))
  action:  run_new_plan();
end

begin tap:
  name: if_time_tap1
  preconds:  (( ((Obs() == Fix4) && (Heading() == N) && (Altitude() ==
Pos) && (Swerve() == False) && (Traffic() == False))))
  action:  turn_left_to_W();
end

begin tap:
  name: if_time_tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == False) && (Obs() == Fix3))))
  action:  obs_set_fix4();
end
```

Plan4: Turn to Final Approach and Initiate Autoland

```
#################################
# plan4.txt
#
# Automatically generated by CIRCA planner.
#
#################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 5;
  NIF_TIME_TAPS=3;
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 15000.000000;
  if_time_tap0 = 15000.000000;
  if_time_tap1 = 15000.000000;
  if_time_tap2 = 9000.000000;
  if_time_tap3 = 10000.000000;
  tap1 = 11000.000000;
  tap2 = 9000.000000;
  tap3 = 8000.000000;
  tap4 = 14000.000000;
end

# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap1, tap0, tap4, tap0, tap2, tap3, tap0 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (( ((Location() == Fix4) && (Swerve() == False) && (Obs()
== Fix0) && (Altitude() == Zero)) ||
 ((Obs() == Fix4) && (Altitude() == Zero))))
  action:  climb_to_altitude();
end

begin tap:
  name: tap2
  preconds:  (( ((Nav_Freq() == Fly) && (Traffic() == False) &&
(Altitude() == Pos) && (Obs() == Fix0))))
  action:  Nav_Freq_land();
end

begin tap:
  name: tap3
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == True))))
  action:  avoid_traffic();
end

begin tap:
  name: tap4
  preconds:  (( ((Nav_Freq() == Land) && (Obs() == Fix0) && (Altitude()
== Pos) && (Traffic() == False) && (Swerve() == True)) ||
 ((Obs() == Fix4) && (Altitude() == Pos) && (Traffic() == False) &&
(Swerve() == True))))
  action:  course_correct();
end

begin tap:
  name: if_time_tap0
  preconds:  (( ((Gear() == Up)) ))
  action:  notify_planner_removed();
end

begin tap:
  name: if_time_tap1
  preconds:  ((Heading() == S) && (Location() == Fix0) && (Obs() ==
Fix0) && (Nav_Freq() == Land))
  action:  run_new_plan();
end

begin tap:
  name: if_time_tap2
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == False) && (Heading() == W))))
  action:  turn_left_to_S();
end

begin tap:
  name: if_time_tap3
  preconds:  (( ((Heading() == S) && (Obs() == Fix4) && (Traffic() ==
False) && (Altitude() == Pos) && (Swerve() == False))))
  action:  obs_set_fix0();
end
```

Plan5: "Gear_up_failure" Contingency Plan

```
#################################
# plan5.txt
#
# Automatically generated by CIRCA planner.
#
#################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 5;
  NIF_TIME_TAPS=2;
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 15000.000000;
  if_time_tap0 = 15000.000000;
  if_time_tap1 = 10000.000000;
  tap1 = 11000.000000;
  tap2 = 9000.000000;
  tap3 = 8000.000000;
  tap4 = 14000.000000;
end
```

```
# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap1, tap0, tap4, tap0, tap2, tap3, tap0 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (( ((Swerve() == False) && (Altitude() == Zero))))
  action:  climb_to_altitude();

begin tap:
  name: tap2
  preconds:  (( ((Location() == Fix4) && (Gear() == Up) && (Obs() ==
Fix0) && (Nav_Freq() == Land) && (Traffic() == False))))
  action:  Nav_Freq_fly();
end

begin tap:
  name: tap3
  preconds:  (( ((Altitude() == Pos) && (Swerve() == False) &&
(Traffic() == True))))
  action:  avoid_traffic();
end

begin tap:
  name: tap4
  preconds:  (( ((Nav_Freq() == Land) && (Obs() == Fix0) && (Altitude()
== Pos) && (Traffic() == False) && (Swerve() == True)) ||
 ((Obs() == Fix4) && (Altitude() == Pos) && (Traffic() == False) &&
(Swerve() == True))))
  action:  course_correct();
end

begin tap:
  name: if_time_tap0
  preconds:  (( ((Gear() == Down)) ))
  action:  notify_planner_deadend();
end

begin tap:
  name: if_time_tap1
  preconds:  (( ((Heading() == S) && (Obs() == Fix0) && (Traffic() ==
False) && (Altitude() == Pos) && (Swerve() == False))))
  action:  obs_set_fix1();
end
```

APPENDIX C

UCAV CIRCA-II KNOWLEDGE BASE AND OUTPUT PLAN FILES

Researchers at the University of Michigan recently participated in a joint UCAV demo with Honeywell Technology Center to illustrate the utility of a probabilistic planner with real-time contingency plan retrieval. In this Appendix, we first include the user-defined text knowledge base for the UCAV, followed by the nominal and contingency [text] plan files produced by the CIRCA-II Planning Subsystem. These plans are both downloaded and stored in the cache, with the nominal plan executing first. Then, when infrared (IR) missile threats are encountered and detected as removed states in plan0, the contingency plan to handle that situation is executed. Ultimately, these plans allow the aircraft to successfully fly its pre-defined route safely even though it is being attacked by a variety of radar and IR missiles.

Knowledge Base

```
########################################################################
#
#  ucav_with_radar.kbase
#
#  CIRCA_C Knowledge base file
#
#  Written:  Ella Atkins
#  Last Modified:  May 16, 1999
#
########################################################################

########################################################################
#
```

```
# Section 1:  Initial States
#
#  Note:  TT is a "special" feature that is always True.
#

begin initial_state:
  Path = Normal;
  Radar_threat = False;
  IR_threat = False;
  Decoy_deployed = False;
  Altitude = High;  # UCAV plan only acts after automatic UCAV takeoff
  Failure = False;
  TT = True;
end

########################################################################
#
#  Section 2:  Subgoal(s)
#

begin subgoal:
  features:      Path = Normal;
  preconditions:  TT = True;
end


########################################################################
#
# Section 3:  Feature WCET's --
#   Set to zero for this test so that TAP wcet = action wcet
#
begin feature_wcets:
  Path = 0.0;
  Radar_threat = 0.0;
  IR_Threat = 0.0;
  Decoy_deployed = 0.0;
  Altitude = 0.0;
  Failure = 0.0;
  TT = 0.0;
end


########################################################################
#
#  Section 4:  Action Transitions
#

begin action:
  name:       blow_chaff
  preconds:   ((f[Radar_threat] == True) && (f[Decoy_deployed] ==
False))
  postconds:  f[Decoy_deployed] = True;
  wcet:       5
end
```

```
begin action:
  name:       deploy_flare_sequence
  preconds:   ((f[IR_threat] == True) && (f[Decoy_deployed] == False))
  postconds:  f[Decoy_deployed] = True;
  wcet:       5
end

begin action:
  name:       begin_radar_evasive
  preconds:   ((f[Path] == Normal) && (f[Decoy_deployed] == True) &&
             (f[Radar_threat] == True))
  postconds:  f[Path] = Evasive;
  wcet:       5
end

begin action:
  name:       begin_IR_evasive
  preconds:   ((f[Path] == Normal) && (f[Decoy_deployed] == True) &&
             (f[IR_threat] == True))
  postconds:  f[Path] = Evasive;
  wcet:       5
end

begin action:
  name:       resume_normal_path
  preconds:   ((f[Path] == Evasive) &&
             !((f[Radar_threat] == True) || (f[IR_threat] == True)))
  postconds:  f[Path] = Normal;
  wcet:       5
end


########################################################################
#
#
#  Section 5:  Temporal Transitions
#

begin temporal:
  name:      radar_threat_tt
  preconds:  ((f[Radar_threat] == False) && (f[Altitude] == High) &&
(f[Path] == Normal))
  postconds: f[Radar_threat] = True;
  prob_func: radar_threat_rate
end

begin temporal:
  name:      IR_threat_tt
  preconds:  ((f[IR_threat] == False) && (f[Altitude] == Low) &&
(f[Path] == Normal))
  postconds: f[IR_threat] = True;
  prob_func: IR_threat_rate
end
```

```
begin temporal:
  name:       evade_radar_missile_tt
  preconds:   ((f[Path] == Evasive) && (f[Decoy_deployed] == True) &&
             (f[Radar_threat] == True))
  postconds:  f[Radar_threat] = False; f[Decoy_deployed] = False;
  prob_func:  evade_radar_missile_rate
end

begin temporal:
  name:       evade_IR_missile_tt
  preconds:   ((f[Path] == Evasive) && (f[Decoy_deployed] == True) &&
             (f[IR_threat] == True))
  postconds:  f[IR_threat] = False; f[Decoy_deployed] = False;
  prob_func:  evade_IR_missile_rate
end

begin temporal:
  name:       swoop_tt
  preconds:   ((f[Radar_threat] == False) &&
             (f[IR_threat] == False) && (f[Altitude] == High))
  postconds:  f[Altitude] = Low;
  prob_func:  swoop_rate
end

begin temporal:
  name:       climb_tt
  preconds:   ((f[Radar_threat] == False) &&
             (f[IR_threat] == False) && (f[Altitude] == Low))
  postconds:  f[Altitude] = High;
  prob_func:  climb_rate
end

#  Temporal Transitions to Failure (TTFs)
#

begin temporal:
  name:      radar_kills_you_tt
  preconds:  ((f[Radar_threat] == True))
  postconds: f[Failure] = True;
  prob_func: radar_kills_you_rate
end

begin temporal:
  name:      IR_kills_you_tt
  preconds:  ((f[IR_threat] == True))
  postconds: f[Failure] = True;
  prob_func: IR_kills_you_rate
end

#######################################################################
#
#  Section 6:  Probability rate function definition
#
```

```
#
begin prob_func:

// evade_radar_missile prob_func -- "reliable" tt with staircase prob
float evade_radar_missile_rate(float time, float dummy)
{
  float min_time = 0.0, max_time=2.0;
  if (time < (min_time-0.001))
    return(0.0);
  else if (time < (max_time - 1.001))
    return(1.0 / (max_time - time));
  else if (time < (max_time - 0.001))
    return(1.0);
  else
    return(0.0);
}

// evade_IR_missile prob_func -- "reliable" tt with staircase prob
float evade_IR_missile_rate(float time, float dummy)
{
  float min_time = 0.0, max_time=2.0;
  if (time < (min_time-0.001))
    return(0.0);
  else if (time < (max_time - 1.001))
    return(1.0 / (max_time - time));
  else if (time < (max_time - 0.001))
    return(1.0);
  else
    return(0.0);
}

// radar_threat prob_func -- "event" transition (can occur at any time)
float radar_threat_rate(float time, float dummy)
{
  return(0.01);
}

// IR_threat prob_func -- "event" transition (can occur at any time)
float IR_threat_rate(float time, float dummy)
{
  return(0.01);
}

// swoop prob_func -- "event" transition (can occur at any time)
float swoop_rate(float time, float dummy)
{
  return(0.0001);
}

// climb prob_func -- staircase "reliable" temporal transition
float climb_rate(float time, float dummy)
{
  float max_time=20.0;  // max-delta time
  if (time < (max_time - 1.001))
```

```
    return(1.0 / (max_time - time));
  else if (time < (max_time - 0.001))
    return(1.0);
  else
    return(0.0);
}

// radar_kills_you prob_func -- ttf
float radar_kills_you_rate(float time, float dummy)
{
  if (time < 45.0)  // before min-delay
      return(0.0);
  else
      return(0.1);  // after min-delay
}

// IR_kills_you prob_func -- ttf
float IR_kills_you_rate(float time, float dummy)
{
  if (time < 35.0)  // before min-delay
      return(0.0);
  else
      return(0.3);  // after min-delay
}

end
#  End of kbase
```

Nominal Plan

```
##################################
# plan0.txt
#
# Automatically generated by CIRCA-II planner.
#
##################################

# Part 1:  Define number of TAPs (if-time TAPs).

begin tap_count:
  NTAPS = 3;
  NIF_TIME_TAPS=2
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 30.000000;
  if_time_tap0 = 30.000000;
  if_time_tap1 = 5.000000;
  tap1 = 5.000000;
  tap2 = 5.000000;
```

```
end

# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap0, tap1, tap0, tap2 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (( ((Decoy_deployed() == False) && (Radar_threat() ==
True))))
  action:  blow_chaff();
end

begin tap:
  name: tap2
  preconds:  (( ((Path() == Normal) && (Decoy_deployed() == True))))
  action:  begin_radar_evasive();
end

begin tap:
  name: if_time_tap0
  preconds:  (( ((IR_threat() == True))))
  action:  notify_planner_removed();
end

begin tap:
  name: if_time_tap1
  preconds:  (( ((Radar_threat() == False) && (Path() == Evasive))))
  action:  resume_normal_path();
end
```

Contingency Plan for IR Missile Threats

```
#################################
# plan1.txt
#
# Automatically generated by CIRCA-II planner.
#
#################################

# Part 1:  Define number of TAPs (if-time TAPs).
```

```
begin tap_count:
  NTAPS = 3;
  NIF_TIME_TAPS=1
end

# Part 2:  Define wcets for TAPs (by tap name -- see below).

begin tap_wcets:
  tap0 = 30.000000;
  if_time_tap0 = 30.000000;
  tap1 = 5.000000;
  tap2 = 5.000000;
end

# Part 3:  Define schedule (by guaranteed TAP name).

begin tap_schedule:
  { tap0, tap1, tap0, tap2 };
end

# Part 4:  Define TAPs.
begin tap:
  name: tap0
  preconds:  (1)
  action:  iftime_server();
end

begin tap:
  name: tap1
  preconds:  (( ((Decoy_deployed() == False) && (Path() == Normal))))
  action:  deploy_flare_sequence();
end

begin tap:
  name: tap2
  preconds:  (( ((Decoy_deployed() == True) && (Path() == Normal))))
  action:  begin_IR_evasive();
end

begin tap:
  name: if_time_tap0
  preconds:  (( ((IR_threat() == False))))
  action:  notify_planner_deadend();
end
```

APPENDIX D
REAL-TIME RESPONSE IN THE PLANNING SUBSYSTEM

The plan cache was incorporated into CIRCA-II to react to dangerous situations as they are encountered. In this dissertation, we present a simple "binary" algorithm in which all dangerous unhandled states are built into contingency plans, and dynamic replanning occurs only for the safe unhandled states. Realistically, we cannot presume hard real-time plan retrieval in all situations if we require cached responses for the exhaustive set of dangerous unhandled states. Thus, in future work, we plan to delve into algorithms for placing bounds on plan development time. Of course, regardless of the algorithm we define, we must still face the time-quality tradeoffs described in [51]. Our goal, however, is to design an algorithm that makes the most "intelligent" tradeoff possible given the state of the world at the time dynamic planning is invoked.

In this appendix, we discuss a possible algorithm for a time-bounded *planner* in CIRCA-II. Incorporating a time-bounded planner is only the first step to bounding real-time control plan development time. We must also eventually bound execution of the other algorithms within the CIRCA-II Planning Subsystem. The real-time community has a collection of dynamic real-time scheduling algorithms we may incorporate [37], but worst-case time for the planner-scheduler iterations required to develop a schedulable plan is more difficult to predict, especially since such operations may require extensive planner backtracking and multiple modifications of the threshold $P_{removed}$ below which states are ignored.[70]

We discussed related work on real-time planning in Chapter II of this dissertation. The two most general approaches the problem are anytime [15] and design-to-time [21] philosophies, each of which may be applied to a variety of algorithms. We propose an approach to limiting planner deliberation time that combines elements from these two methods. As shown in Figure D-1, upon receipt of a state for which plans must be developed online, the planner first computes available deliberation time. This quantity is used in a design-to-time fashion to set up CIRCA-II planning parameters. Finally, the planner executes using a best-first search until deliberation time *(t_{delib})* expires. Each of these procedures is described in more detail below.

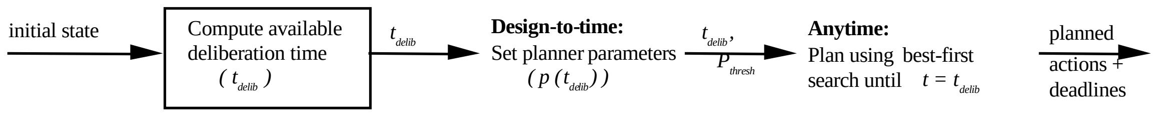

initial state
Compute available deliberation time (t_{delib})
t_{delib}
Design-to-time:
Set planner parameters ($p(t_{delib})$)
t_{delib}, P_{thresh}
Anytime:
Plan using best-first search until $t = t_{delib}$
planned actions + deadlines

Figure D-1. Proposed Algorithm for Limiting Planner Deliberation Time.

To compute the planner deliberation time t_{delib}, we plan to use the planner's initial "unhandled" state (fed back from the Plan Execution Subsystem) to quickly compute an initial estimate of a planning time limit, then potentially modify this estimate based on environment changes during planning. Since CIRCA's main goal is always maintaining system safety, the limiting factor for deliberation time is how long the system will be guaranteed to remain safe executing the currently executing plan. To estimate the time to

[70] One possible method for bounding planning subsystem execution is to redesign the overall algorithm such that the scheduler is called each time a new guaranteed action is selected. In this manner, any schedulability violation will be immediately identified and acted on before the planner completes its state expansion process. We have not explored this avenue sufficiently to provide more details in this dissertation.

failure from this initial state, CIRCA may perform a lookahead state-space projection from this state, using all *tts* in the knowledge base as well as the currently executing plan to specify action choices and timings. The nearest TTF identified with probability above P_{thresh} corresponds with the deliberation time limit.[71]

Next, we wish to adjust planner parameters so that it expected time of completion will be just under t_{delib}. We have no obvious parameters to control prior to planning except for the probability threshold values P_{thresh} and $P_{removed}$ (see definitions in Chapter IV). We are already modifying $P_{removed}$ dynamically to adjust probabilistic failure-avoidance guarantees as required for plan scheduling. Additionally, $P_{removed}$ does not affect plan development at each step, but instead affects the *number* of states the planner must expand during its best-first search. Thus, we propose adjusting P_{thresh} in accordance with design-to-time limitations, although we have not yet computed even an approximate algebraic relationship between P_{thresh} and t_{delib}.[72]

In CIRCA-II, state expansion occurs in decreasing-probability order until all states remaining to expand have probability less than $P_{removed}$. In the original version of CIRCA, search proceeded depth-first, so there was no guarantee that the resulting goal path was any more desirable than other possible goal paths. Research described in this dissertation discusses the basic conversion to best-first search based solely on state probability estimates. However, in this work, "best" is based completely on state probability, with state expansion occurring in decreasing order of state probability.

---

[71] To compute the time to the nearest *ttf*, we must store the cumulative minimum delay (i.e., sum of all transition $min\Delta$ values given the currently-executing plan and its action deadlines) leading down the path to this *ttf*.

[72] The mapping between P_{thresh} and planning time is not obvious and needs further study. This choice is proposed because augmenting P_{thresh} makes failure states more likely, thus if time permits P_{thresh} should be minimized. However, since each "failure" state is absorbing, an increased chance in reaching failure means new non-failure states with probability above $P_{removed}$.

The planner may combine its knowledge about probabilities, temporal delays, and proximity to failure to achieve a better measure than probabilities alone to control the best-first search. State expansion may be ordered by decreasing utility $u(s_k)$, as shown in Equation (D-1), where $P(s_k)$ is the probability of visiting state s_k (see Chapter IV), $t_{min}(s_k)$ is the minimum time before the system can reach state s_k, $P_{failure}(s_k,n)$ is the probability of any failure state occurring in n (or fewer) time steps from state s_k. The constants a, b, c, and n (if constant) are as yet undetermined. By expanding states in this order, we will plan for the most "important" states, not just the most likely states, achieving a balance between state probability, system safety (i.e., prioritizing expansion to handle states that can reach failure), and the time horizon considered by the planner (i.e., near-term states are handled; far-term states will be handled by subsequent plans).

$$u\,(s_k) = a * P(s_k) + (b \,/\, t_{min}(s_k)) + c * P_{failure}(s_k,\, n) \qquad \text{(D-1)}$$

APPENDIX E

TACKLING THE MULTIPLE CYCLE, MULTIPLE *DTTF* CHALLENGE

Figure E-1 depicts a state-space example that is not properly handled by the probabilistic planner within CIRCA-II. This figure represents a valid plan because all temporal transitions have sufficient delays to be preempted with guaranteed actions. However, this plan will not be found because of the cycle leading from s_3 back to s_1. In this appendix, we first describe this problem and then outline a possible solution approach for this class of problems within both the CIRCA and CIRCA-II planners.

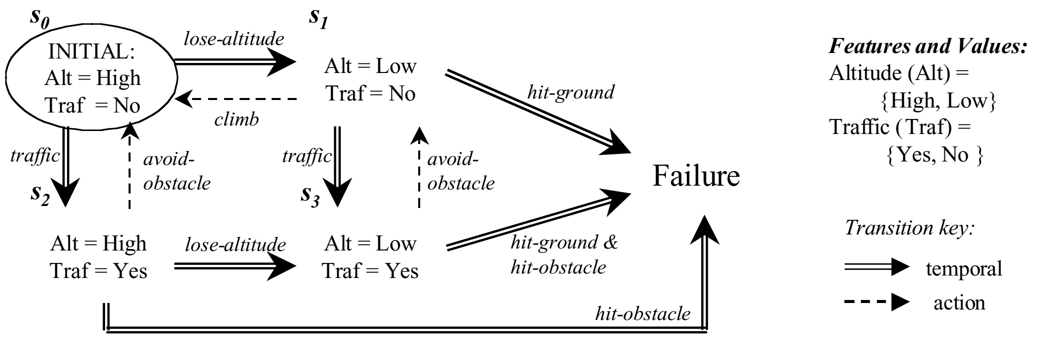


Figure E-1: State-space Example with Multiple Cycles and Dependent *ttfs*.

For the example shown in Figure E-1, CIRCA-II begins by selecting no action for state s_0. Then, when first expanding s_1, CIRCA-II selects *climb* to avoid *hit-ground* and can guarantee it will preempt *hit-ground* since it is not yet a dependent temporal transition. However, upon expanding s_3, CIRCA-II must select a preemptive action that forms a cycle back to s_1. CIRCA-II then re-expands s_1 and determines *hit-ground* is dependent and may occur more quickly than previously thought (since the initial delay between transitioning to the low altitude and crashing may have evaporated during the s_1-

s_3- s_1 cycle traversal). For this example, after one cycle traversal, the system still has time to execute *climb* and return to safe state s_0. However, if *traffic* occurs before *climb* completes, the system will again be thrown into s_3 and the remaining safety margin before *hit-ground* can occur disappears before the s_3->s_1 transition again completes, thus the planner fails. We have previously said there is a non-zero delay between the time a transition is first active and when it could occur. From the s_0-s_1 path, this delay may have evaporated since *traffic* is also active in s_0. However, along the cyclic path returning from s_3, this delay is again available since *traffic* is not active in s_3, thus the guaranteed action *climb* will then preempt both *hit-ground* and *traffic* and return safely to state s_0.

This problem surfaces because both CIRCA and CIRCA-II are combining the effects of all parent paths in order to minimize the state-space size. CIRCA-II incorporates all parent path effects using the weighted-average algorithms described in Chapter IV for both state probability and *ttf* preemption timing computations while CIRCA presumes worst-case properties (e.g., smallest *ttf* minimum delays; maximum preemptive action delays) every state transition, including *dtts*.

To solve state-spaces similar to that in Figure E-1, CIRCA and CIRCA-II must distinguish between the effects of entering state s_1 from parent s_0 versus s_3 because these two paths result in different temporal constraints that ultimately give the appearance that preempting *hit-ground* from s_1 is impossible. When CIRCA-II identifies such a situation in which individual parent states impose more restrictive constraints *together* than would be imposed due to each individual parent, we propose that the planner mark this state as a candidate for *splitting* into two states with a special feature used to identify the parent(s) for that state. Then, if timing constraints appear to be impossible to satisfy,[73] the state

[73] Constraints will be impossible to satisfy *immediately* for this example, but could require relaxation *subsequently* if scheduling difficulties arise.

would actually be split, with each of the split states requiring only preemption of its parent, not the conglomerate set of parents into that original state.

For this example, splitting s_1 will allow the planner to accurately determine that it can guarantee *ttf* preemption using the one action that leads from s_1 back to s_0. However, in general, the two states resulting from a split may require different actions. This is not a problem for state-expansion because the parent reference feature value specified for a split state acts as any other feature. However, during TAP development and plan execution, the only way to detect the value for the parent feature (and thereby execute the appropriate response) is to have sensed and stored the previous state, a function that is not straightforward to implement given the current TAP plan structure. We have not yet given careful thought to the tradeoffs involved or the mechanisms required for storing any previous state history within the CIRCA-II Plan Execution Subsystem.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin, QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control, in: *Proceedings of the Third IEEE Real-time Technology and Applications Symposium*, Montreal, Canada (1997) 228-238.

[2] T. F. Abdelzaher, A. Shaikh, F. Jahanian, and K. G. Shin, RTCAST: Lightweight Multicast for Real-time Process Groups, in: *Proceedings of the Second IEEE Real-time Technology and Applications Symposium*, Boston, Massachusetts (1996).

[3] T. F. Abdelzaher and K. G. Shin, Optimal Combined Task and Message Scheduling in Distributed Real-time Systems, in: *Proceedings of the Sixteenth IEEE Real-time Systems Symposium,* Pisa, Italy, (1995) 162-171.

[4] J. Arnold, Dynamic Backtracking and Goal Decomposition in CIRCA-II, Technical Report *in progress*, University of Michigan.

[5] E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee, Planning and Resource Allocation for Hard Real-time, Fault-Tolerant Plan Execution, in: *Proceedings of the Third International Conference on Autonomous Agents,* Seattle, Washington (1999) 244-251.

[6] E. M. Atkins, R. H. Miller, T. VanPelt, K. D. Shaw, W. B. Ribbens, P. D. Washabaugh, and D. S. Bernstein, Solus: An Autonomous Aircraft for Flight Control and Trajectory Planning Research, in: *Proceedings of the American Control Conference,* Philadelphia, Pennsylvania, 2 (1998) 689-693.

[7] E. M. Atkins, E. H. Durfee, and K. G. Shin, Detecting and Reacting to Unplanned-for World States, in: *Proceedings of the Fourteenth National Conference on Artificial Intelligence,* Providence, Rhode Island, (1997) 571-576.

[8] E. M. Atkins, E. H. Durfee, and K. G. Shin, Plan Development in CIRCA using Local Probabilistic Models, in: *Uncertainty in Artificial Intelligence: Proceedings of the Twelfth Conference,* Portland, Oregon, (1996) 49-56.

[9] R. P. Bonasso, H. J. Antonisse, and M. G. Slack, Reactive Robot System for Find and Fetch Tasks in an Outdoor Environment, in: *Proceedings of the Tenth National Conference on Artificial Intelligence,* San Jose, California, (1992) 801-808.

[10] C. Boutilier and R. Dearden, Using Abstractions for Decision-Theoretic Planning with Time Constraints, in: *Proceedings of the Twelfth National Conference on Artificial Intelligence,* Seattle, Washington, (1994) 1016-1022.

[11] C. Boutilier, T. Dean, and S. Hanks, Decision-Theoretic Planning: Structural Assumptions and Computational Leverage, to appear in*: Journal of Artificial Intelligence Research (JAIR).*

[12] D. J. Brudnicki and D. B. Kirk, Trajectory Modeling for Automated En Route Air Traffic Control (AERA), *Proceedings of the American Control Conference* (1995) 3425-3429.

[13] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman, Acting Optimally in Partially Observable Stochastic Domains, *Proceedings of the Twelfth National Conference on Artificial Intelligence,* Seattle, Washington, (1994).

[14] T. Dean, R. Givan, and K. Kim, Solving Stochastic Planning Problems with Large State and Action Spaces, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, Pittsburgh, Pennsylvania, (1998).

[15] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson, Planning under Time Constraints in Stochastic Domains, *Artificial Intelligence* 76(1-2) (1995) 35-74.

[16] J. L. Devore, *Probability and Statistics for Engineering and the Sciences*, Second Edition (Wadsworth, Inc., Belmont, California 1987).

[17] E. Gat, Three-Layer Architectures, in: *Artificial Intelligence and Mobile Robotics* (MIT Press, Cambridge, Massachusetts 1998).

[18] R. E. Fikes and N. J. Nilsson, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence* 2(3-4)(1971) 189-208.

[19] R. J. Firby, An Investigation into Reactive Planning in Complex Domains, in: *Proceedings of the National Conference on Artificial Intelligence* (1987) 202-206.

[20] A. Garvey, K. Decker, and V. Lesser, A Negotiation-based Interface Between a Real-time Scheduler and a Decision-Maker. *Technical Report 94-08,* University of Massachusetts Department of Computer Science (1994).

[21] A. Garvey and V. Lesser, Design-to-time real-time scheduling, *IEEE Transactions on Systems, Man and Cybernetics* 23(6)(1993) 1491-1502.

[22] E. Gat, Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Mobil Robots, *SIGART Bulletin* 2(1991) 70-74.

[23] M. L. Ginsberg, Universal Planning: An (Almost) Universally Bad Idea, *AI Magazine*, 10(4)(1989).

[24] M. L. Ginsberg, Dynamic Backtracking, *Journal of Artificial Intelligence Research* 1(1993) 25-46.

[25] R. Goldman, D. Musliner, K. Krebsbach, and M. Boddy, Dynamic Abstraction Planning, in: *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Providence, Rhode Island, (1997) 680-686.

[26] R. Goldman, M. Pelican, and D. Musliner, Hard Real-time Mode Logic Synthesis for Hybrid Control: A CIRCA-based Approach, *Working Notes of the AAAI Symposium on Hybrid Control* (1999).

[27] L. Greenwald and T. Dean, Solving Time-Critical Decision-Making Problems with Predictable Computational Demands, in: *Proceedings of the Second International Conference on AI Planning Systems* (1994).

[28] K. Hammond, *Case-Based Planning: Viewing Planning as a Memory Task*, (Academic Press, New York 1989).

[29] S. Hanks, and D. McDermott, Modeling a Dynamic and Uncertain World: Symbolic and Probabilistic Reasoning About Change, *Artificial Intelligence* 66(1)(1994) 1-55.

[30] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, SPUDD: Stochastic Planning using Decision Diagrams, to appear in the *Proceedings of the Uncertainty in Artificial Intelligence (UAI) Conference* (1999).

[31] E. Horvitz and M. Barry, Display of Information for Time-Critical Decision Making, *Uncertainty in Artificial Intelligence: Proceedings of the Eleventh Conference* (August 1995).

[32] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert, PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots, in: *Proceedings of the Thirteenth IEEE International Conference on Robotics and Automation,* Minneapolis, Minnesota, 1 (1996) 43-49.

[33] F. F. Ingrand and M. P. Georgeff, "Managing Deliberation and Reasoning in Real-Time AI Systems," in *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, (November 1990) 284-291.

[34] R. Jones, J. Laird, and P. Nielsen, Automated Intelligent Pilots for Combat Flight Simulation, in: *Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence*, Madison, Wisconsin, (1998) 1047-1054.

[35] J. Juang, *Applied System Identification*, (Prentice-Hall, Englewood Cliffs, New Jersey 1994).

[36] J. G. Kemeny, and J. L. Snell, *Finite Markov Chains* (1960).

[37] C. M. Krishna and K. G. Shin, *Real-Time Systems,* (McGraw-Hill, New York 1997).

[38] B. C. Kuo, *Automatic Control Systems*, sixth edition (Prentice-Hall, Englewood Cliffs, New Jersey 1991).

[39] N. Kushmerick, S. Hanks, and D. Weld, An Algorithm for Probabilistic Planning, in: *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994) 1073-1078.

[40] J. E. Laird, A. Newell, and P. S. Rosenbloom, SOAR: An Architecture for General Intelligence, *Artificial Intelligence*, 33(1)(1987) 1-64.

[41] D. A. Lawrence and W. J. Rugh, Gain Scheduling Dynamic Linear Controllers for a Nonlinear Plant, *Automatica*, 31(3)(March 1995) 381-390.

[42] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, (Prentice-Hall, Englewood Cliffs, New Jersey 1981).

[43] H. Li, A Minimal Temporally-Dependent Probabilistic World Model Using Temporally-Independent States, *Directed Study Report*, University of Michigan (May 1999).

[44] S. Liden, The Evolution of Flight Management Systems, *Proceedings of the 1994 IEEE/AIAA Thirteenth Digital Avionics Systems Conference*, (1995) 157-169.

[45] M. L. Littman, T. L. Dean, and L. P. Kaelbling, On the Complexity of Solving Markov Decision Problems, in: *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (1995).

[46] D. G. Luenberger, *Introduction to Dynamic Systems: Theory, Models and Applications.* (Wiley, New York 1978).

[47] C. B. McVey, E. M. Atkins, E. H. Durfee, and K. G. Shin, Development of Iterative Scheduler to Planner Feedback, in: *Proceedings of the International Joint Conference on Artificial Intelligence,* (1997) 1267-1272.

[48] R. H. Miller and W. B. Ribbens, The Effects of Icing on the Longitudinal Dynamics of an Icing Research Aircraft, Number 99-0636 in *37th Aerospace Sciences*, AIAA (American Institute of Aeronautics and Astronautics) (January 1999).

[49] R. H. Miller and W. B. Ribbens, Detection of the Loss of Elevator Effectiveness due to Aircraft Icing, Number 99-0637 in *37th Aerospace Sciences*, AIAA (American Institute of Aeronautics and Astronautics) (January 1999).

[50] N. Muscettola, P. Nayak, B. Pell, B. Williams, Remote Agent: To Boldly Go where no AI System has gone Before, *Artificial Intelligence* 103(1/2)(August 1998).

[51] D. J. Musliner, E. H. Durfee, and K. G. Shin, World Modeling for the Dynamic Construction of Real-Time Control Plans, *Artificial Intelligence,* 74(1) (1995) 83-127.

[52] D. J. Musliner, Scheduling Issues Arising from Automated Real-Time System Design. *University of Maryland Department of Computer Science Technical Report CS-TR 3364*, UMIACS-TR-94-118.

[53] D. J. Musliner, CIRCA: The Cooperative Intelligent Real-Time Control Architecture. *University of Michigan Department of EECS, CSE Division Technical Report*, CSE-TR-175-93.

[54] D. J. Musliner, Predictive Sufficiency and the Use of Stored Internal State, in: *Proceedings of the Conference on Intelligent Robotics in Field, Factory, Service, and Space*, Houston TX (1994) 298-305.

[55] B. Pell, D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, and B. Williams, Autonomous Spacecraft Agent Prototype, *Autonomous Robots* 5(1) (1998) 29-52.

[56] T. Peng, K. G. Shin, and T. F. Abdelzaher, Assignment and Scheduling of Communicating Periodic Tasks in Distributed Real-time Systems, *IEEE Transactions on Parallel and Distributed Systems,* 8(12) (1997).

[57] J. R. Quinlan, Induction of Decision Trees, *Machine Learning* 1(1986) 81-106.

[58] R. Rainey, *ACM: The Aerial Combat Simulation for X11*, (http://www.websimulations.com).

[59] O. R. Reynolds, H. Pachter, and C. H. Houpis, Full Envelope Flight Control System Design using Qualitative Feedback Theory, *Journal of Guidance, Control, and Dynamics*, 29(1)(1996) 23-29.

[60] J. Rowland, *Linear Control Systems: Modeling, Analysis, and Design* (Wiley, New York 1986).

[61] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (Prentice-Hall, New Jersey 1995).

[62] S. Russell and D. Subramanian, Provably Bounded-Optimal Agents, *Journal of Artificial Intelligence Research* 2(1995) 575-609.

[63] M. J. Schoppers, Universal Plans for Reactive Robots in Unpredictable Environments, in: *Proceedings of the International Joint Conference on Artificial Intelligence,* (1987) 1039-1046.

[64] J. Schreur, B737 Flight Management Computer Flight Plan Trajectory Computation and Analysis, *Proceedings of the American Control Conference,* (1995) 3419-3429.

[65] J. Shapiro, Implementing CIRCA-II on the QNX Real-time Operating System, Technical Report *in progress*, University of Michigan.

[66] T. Shepard and M. Gagne, A Pre-Run-Time Scheduling Algorithm for Hard Real-time Systems, *IEEE Transactions on Software Engineering*, 17(7)(July 1991) 669-677.

[67] R. A. Slattery, Terminal Area Trajectory Synthesis for Air Traffic Control Automation, *Proceedings of the American Control Conference,* (June 1995) 1206-1210.

[68] M. J. Stefik, *Introduction to Knowledge Systems.* (Morgan Kaufmann, San Francisco, California 1995).

[69] M. Tambe, J. Adibi, Y. Alonaizon, A. Erdem, G. Kaminka, S. Marsella, and I. Muslea, Building Agent Teams using an Explicit Teamwork Model and Learning, to appear in *Artificial Intelligence* (1999).

[70] T. Wagner, A. Garvey, V. Lesser, Criteria-Directed Task Scheduling, *International Journal of Approximate Reasoning* 19(1/2)(1998) 91-118.

[71] E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley, Planning and Reacting in Uncertain and Dynamic Environments, *Journal of Experimental and Theoretical AI*, 7(1)(1995) 197-227.

[72] J. Xu and D. L. Parnas, Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations, *IEEE Transactions on Software Engineering*, SE-16(3)(March 1990) 360-369.

[73] J. Xu, Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations, *IEEE Transactions on Parallel and Distributed Systems*, 8(12)(December 1997).

[74] S. Zilberstein and S. Russell, Optimal Composition of Real-time Systems, *Artificial Intelligence* 82(1/2)(April 1996) 181-213.

[75] S. Zilberstein, Using Anytime Algorithms in Intelligent Systems, *AI Magazine* 17(3)(1996) 73-83.