
Autonomous Agents and Multi-Agent Systems, 4, 57–78, 2001

Planning and Resource Allocation for Hard Real-time, Fault-Tolerant Plan Execution

ELLA M. ATKINS atkins@eng.umd.edu
University of Maryland

TAREK F. ABDELZAHER zaher@eecs.umich.edu
University of Virginia

KANG G. SHIN AND EDMUND H. DURFEE {kgshin, durfee}@umich.edu
University of Michigan

Abstract. We describe the interface between a real-time resource allocation system with an AI planner in order to create fault-tolerant plans that are guaranteed to execute in hard real-time. The planner specifies the task set and all execution deadlines required to ensure system safety, then the resource utilization. A new interface module combines information from planning and resource allocation to enforce development of plans feasible for execution during a variety of internal system faults. Plans that over-utilize any system resource trigger feedback to the planner, which then searches for an alternate plan. A valid plan for each specified fault, including the nominal no-fault situation, is stored in a plan cache for subsequent real-time execution. We situate this work in the context of CIRCA, the Cooperative Intelligent Real-time Control Architecture, which focuses on developing and scheduling plans that make hard real-time safety guarantees, and provide an example of an autonomous aircraft agent to illustrate how our planner-resource allocation interface improves CIRCA performance.

Keywords: AI architectures, planning, real-time scheduling, fault-tolerance

1. Introduction

AI planners have demonstrated utility for converting domain knowledge into situationally-relevant plans of action, but execution of these plans cannot generally guarantee hard real-time response. Planners and plan-execution systems have been extensively tested on problems such as mobile robot control and various dynamic system simulations, but the majority of architectures used today employ a "best-effort" strategy. As such, these systems are only appropriate for soft real-time domains in which missing a task deadline does not cause total system failure.

To control mobile robots, for example, soft real-time plan execution succeeds because the robot can slow down to allow increased reaction times, or even stop moving should the route become too hazardous. For more "unstable" applications such as fully-automated aircraft flight, hard real-time response is required, and fault-tolerance is mandated. Moreover, to achieve complete automation of such a system, some form of planner may be desired, particularly for selecting reactions to anomalous or emergency situations.

Violating response timing constraints in safety-critical systems may be catastrophic. The real-time research community focuses its efforts on resource allocation and scheduling algorithms to provide hard real-time execution guarantees. The main goal of such algorithms is often the efficient utilization of resources such as multiprocessor networks and communication channels. To date, these real-time algorithms have been tested by presuming the existence of complete and inflexible plans of action, including specific task constraints such as execution deadlines. Developing explicit mechanisms for degrading system performance if resource shortage does not allow all constraints to be met is a major real-time research issue.

In this paper, we describe the interface of a real-time resource allocator with an AI planner to automatically create fault-tolerant plans that are guaranteed to execute in hard real-time. The planner produces an initial plan and task constraint set required for failure avoidance. This plan is scheduled by the resource allocator for the nominal "no-fault case" as well as for each specified "internal" fault condition, such as a processor or communication channel failure. If any resource is over-utilized, the most costly task is determined using a heuristic that combines utilization information with task priority (or value) assigned by the planner. The costly task is fed back to the planner, which invokes dynamic backtracking to replace this task, or, if required, ignores unlikely events to generate a more schedulable task set. This combination of planning and resource allocation takes the best elements from both technologies: plans are created automatically and are adaptable, while plan execution is guaranteed to be tolerant to potential faults from a user-specified list and capable of meeting hard real-time constraints.

As a testbed for this combination, we have augmented the Cooperative Intelligent Real-time Control Architecture (CIRCA) [19] and have entitled the updated version CIRCA-II. Originally designed to allow real-time plan execution guarantees, CIRCA uses a planner and a uniprocessor scheduler to build and schedule its plans. We have recently focused on the interface between planner and scheduler [17]. We improve upon this work by adding fault-tolerance and the capability to reason about multiple resource classes and instances of each class, so that all aspects relevant to plan execution are explicitly considered during the planning phase.

This paper begins by describing CIRCA and its evolution into CIRCA-II, followed by a description of the real-time resource allocation algorithms that provide the basis for our work. Next, we present a heuristic to combine pertinent information from planning and resource allocation modules. The resulting "bottleneck" task information is used to guide planner backtracking when scheduling conflicts arise. We then describe an algorithm which incorporates this heuristic and a fault condition list to develop a set of fault-tolerant plans which will execute with hard real-time safety guarantees. We illustrate the utility of our algorithms with a simple example of an autonomous aircraft agent that must be tolerant to a single-processor failure during plan execution, then describe related work in the development of planning and plan-execution agent architectures. We conclude with a summary and discussion of future work required to further bridge the gap between the planning and hard real-time research communities.

2. Building plans for hard real-time execution

We selected the Cooperative Intelligent Real-time Control Architecture (CIRCA) for this research due to its focus on hard real-time plan execution and because its modular components were ideal for interfacing an existing planner with a standard real-time resource allocation algorithm. In this section, we briefly review CIRCA as it has appeared in previous work. Next, we describe CIRCA-II, specifically focusing on its methods for efficiently converging on schedulable plans and for tolerating computational resource faults when plans are executed on a multi-resource platform. Although the CIRCA-II state-space planner is similar to the CIRCA planner, we have modified its backtracking to display a bias toward the production of *schedulable* real-time plans. We close this section with a discussion of CIRCA-II replanning when scheduling fails.

2.1. CIRCA background

Figure 1 shows the CIRCA architecture originally developed by Musliner et al. [19]. The CIRCA domain knowledge base specifies how system state may change via a set of action and temporal state transitions, and contains a set of subgoals which, when achieved in order, enable the system to reach its final goal. During planning, the world model is created incrementally based on initial state(s) and all available transitions. The planner builds a state-transition network from initial to goal states and selects an action (if any) for each state based on the relative gain from performing the action. The planner backtracks if the action selected for any state does not ultimately help achieve the goal or if the system cannot be guaranteed to avoid failure. CIRCA's planner minimizes memory and time usage by expanding only states produced by transitions from initial states or their descendants, and includes a probability model [5] which promotes a best-first state-space search as well as limiting search size via removal of "highly improbable" states [7]. Planning terminates when the goal has been reached while avoiding failure states.

During plan construction, action transition timing constraints are determined such that the system will be guaranteed to avoid all *temporal transitions to failure (TTFs)*, any one of which would be sufficient to cause catastrophic system failure. The CIRCA temporal model allows computation of a minimum delay before each *TTF* can occur; then the deadline for each pre-emptive action is set to this minimum delay. After building each plan, CIRCA explicitly schedules all pre-emptive actions (tasks) such that the plan is guaranteed to meet all such deadlines, thus guaranteeing failure avoidance.

In previous CIRCA work, completed plans were scheduled using a uniprocessor scheduler. If scheduling was successful, the plan was downloaded and executed. Otherwise, the planner backtracked to select a different set of actions which would hopefully be easier to schedule.

After a schedulable plan is developed, it executes on CIRCA's real-time plan executor. Figure 2 shows the contents of a typical CIRCA plan. Tasks, built as TAPs (Test-Action Pairs) by the CIRCA planner, are divided into two classes: *guaranteed*

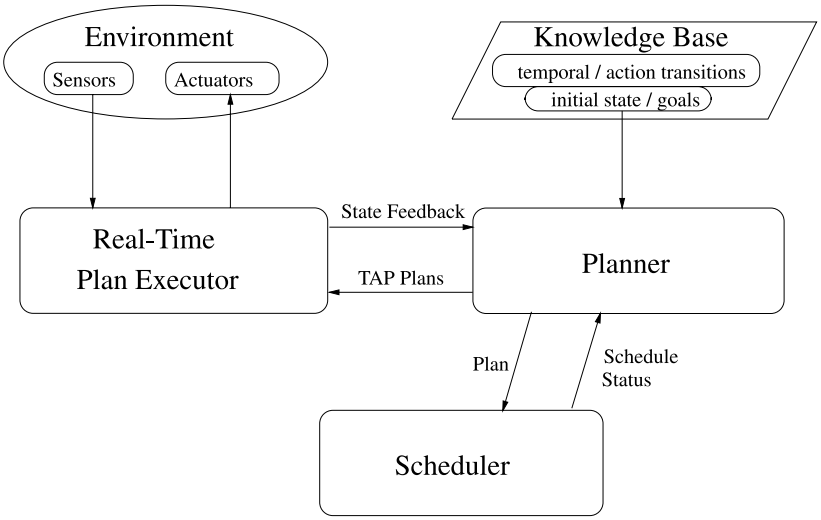

Figure 1. Original CIRCA.

and *best-effort*. Guaranteed tasks have hard real-time maximum separation constraints to avoid TTFs, while best-effort tasks need only execute for goal achievement, thus are acceptable when cast in a "soft real-time" framework. Each task consists of multiple threads (or "modules," the term we will use throughout this paper) which perform the tests required to determine if the action should be executed and then to execute the action, if required.

The CIRCA plan executor previously required a single-processor platform in which all actions were constrained to execute entirely on one dedicated processor, since delays were contained within the static worst-case execution times (wcets) used by the uniprocessor scheduler. Also, the scheduler considered only that one processor need be scheduled, so neither multiple resource instances (e.g., multiple

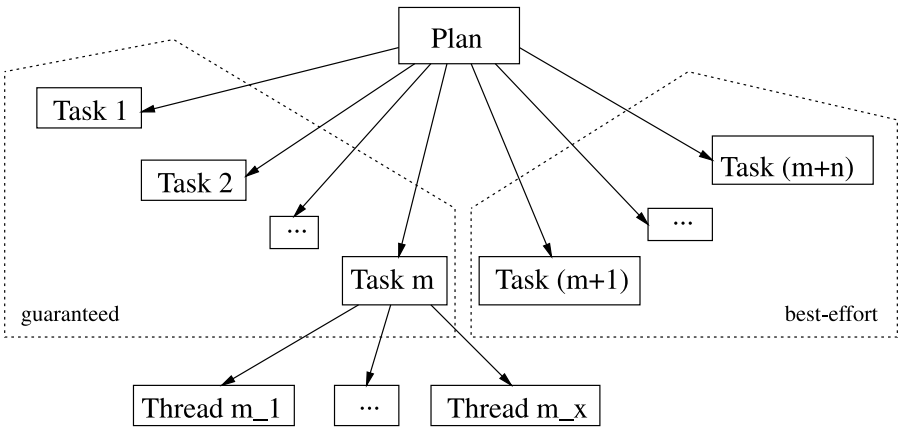

Figure 2. CIRCA plan composition.

processors) nor multiple resource classes (e.g., processors and communication channels) could be considered during scheduling. As a result, not only was the plan-execution platform inflexible, but there was no possibility for computational system fault-tolerance since multiple copies of a resource could not exist.

2.2. CIRCA-II architecture

In this paper, we extend CIRCA to consider multiple resources during plan scheduling and exhibit limited tolerance to resource failures (i.e., "internal faults") during plan execution. Figure 3 shows the CIRCA-II architecture, which includes the modifications required to gain these new capabilities. The CIRCA-II knowledge base and planner are very similar to those from CIRCA. Plans are created and action timing constraints are computed as before in [19] and [5]. For each planned task $T_i \in T_{total}$, where T_{total} contains all tasks in the plan, the planner outputs the triplet (g_i, P_i, V_i). g_i is the "guarantee flag" that indicates whether task T_i is guaranteed $(g_i = 1)$ or best-effort $(g_i = 0)$. P_i is the period of T_i required to preempt TTFs when $g_i = 1$.[1] Finally, V_i is the "priority" value of task T_i and is described below in Section 4.

The planner passes (g_i, P_i, V_i) for all $T_i \in T_{total}$ to the planner-resource allocation interface module, henceforth referenced simply as the "interface." This interface invokes the resource allocation analyzer to schedule all tasks $(T_i \in T_{total}, g_i = 1)$ for each fault to be tolerated. Replacing CIRCA's uniprocessor scheduler, the resource allocation analyzer accounts for multiple plan-execution resources, and also contains the fault library. When a "good" (schedulable) plan is found for each fault, it is downloaded to the plan cache, where it waits until required for execution.

The plan dispatcher/cache modules shown in Figure 3 allow offline storage of plans that may be required to handle either "unexpected" environmental situations [4] or computational resource faults. In this paper, we describe the production of plan sets for the cache that allow CIRCA-II to respond in real-time to computational system faults by retrieving a plan built specifically for that fault (or fault set). Details of the current Plan Execution Subsystem design and implementation are provided in [4]. Due to the complexity of the algorithms required for real-time fault detection and run-time scheduling, significant discussion of the multi-resource Real-time Plan Executor is reserved for future publication.[2]

In subsequent sections, we will discuss the details of the resource allocation, feedback generation, and fault-tolerant plan production algorithms. First, however, we focus on the backtracking procedure that enables the CIRCA-II planner to actually utilize this feedback.

2.3. Converging on a "Schedulable" plan in CIRCA-II

Whenever a proposed plan is deemed unschedulable, the interface module heuristically computes the most costly action (task $T_{bottleneck}$), which it then feeds back to the planner. The planner attempts to remove or replace this action via dynamic

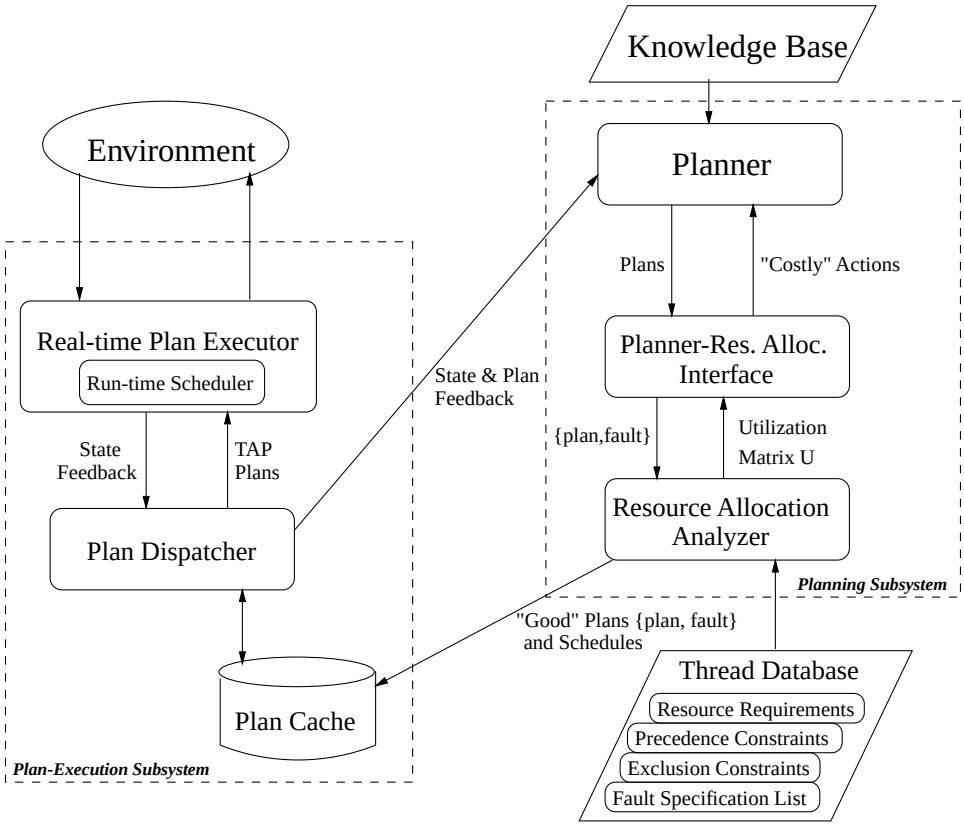

Figure 3. CIRCA-II architecture.

backtracking [13] to each state in which the $T_{bottleneck}$ action was both chosen and guaranteed ($g_i = 1$). Figure 4 describes the current algorithm used by the CIRCA-II planner during backtracking.[3] Ideally, the action from $T_{bottleneck}$ can be modified (i.e., max-period P_i increased) or replaced in every state s_k to which the planner backtracks. If the plan produced after dynamic backtracking over all s_k is potentially *less restrictive* (see definition below) than all previous plans, it is proposed for scheduling. Otherwise, the planner continues to search for a new plan via a combination of alternative backtracking methods.

Definition: We define $plan_1$ as potentially *less restrictive* than $plan_2$ if one of the following criteria is met:

1. All tasks $T_{1i} \in plan_1$ form a proper subset of the tasks $T_{2j} \in plan_2$.
2. $\exists i, j((T_{1i} = T_{2j}) and (P_{1i} > P_{2j}))$, where P_{ki} represents the period assigned to task T_{ki} in $plan_k$.

As shown in Figure 4, our dynamic backtracking algorithm cannot alone guarantee the production of a schedulable plan. Thus we have combined dynamic with

1. Initialize $plan_{pj}$ to unschedulable plan $plan_{pi}$ minus bottleneck task $T_{bottleneck}$
2. Perform dynamic backtracking to each reachable state s_k in $plan_{pi}$ requiring $T_{bottleneck}$ for TTF preemption.
   - If ($\exists T_k s.t.(T_k \in plan_{pj})$ and $(P_k \geq P_{pj})$ and (T_k preempts all s_k TTFs), where P_k is the periodic requirement for task T_k in s_k and P_{pj} is the period already assigned to T_k in $plan_{pj}$,
     Select T_k for s_k; $plan_{pj}$ remains the same.
   - Else if ($\exists T_k s.t.(T_k \neq T_{bottleneck})$ and (T_k preempts all TTFs out of s_k) and ($plan_{pj}$ with (T_k, P_k) is a potentially less-restrictive alternative plan)) then
     Select T_j for s_k; add (T_j, P_j) to $plan_{pj}$.
   - Else if ($plan_{pj}$) with $(T_{bottleneck}, P_k)$ is a potentially less-restrictive alternative plan) then
     Select $T_{bottleneck}$ for s_k; add $(T_{bottleneck}, P_k)$ to $plan_{pj}$.
   - Else
     Dynamic backtracking fails; break out of s_k loop.
   - Propagate the effects of s_k task modification throughout the state-space; modify $plan_{pj}$ to guarantee TTF preemption in all downstream states. If the resulting $plan_{pj}$ cannot be less-restrictive than the unschedulable $plan_{pi}$, return to beginning of Step 2 and select a different T_k for this s_k.
   - Return to beginning of Step 2 for the next s_k.
3. If dynamic backtracking has failed for any s_k, reset $plan_{pj}$ to $plan_{pi}$ and perform chronological backtracking until finding a less-restrictive alternative plan or failing.
4. If chronological backtracking has failed, increment probability threshold P_{thresh} below which improbable states are ignored and restart the planner.

Figure 4. CIRCA-II planner backtracking.

other backtracking techniques, the choice of which derives from the following three alternatives:

Method 1: Perform chronological backtracking each time dynamic backtracking fails to produce an alternate plan. This method is incapable of incorporating interface feedback and reverts to the original CIRCA approach. It is, however, capable of quickly proposing alternative plans.

Method 2: Request the next-most costly task $T2_{bottleneck}$ from the interface. Perform dynamic backtracking with $T2_{bottleneck}$, and continue (e.g., with the third most costly task, ...) until dynamic backtracking has been performed on each individual task. This method is desirable in that all available interface feedback is utilized.

However, backtracking is far from exhaustive because each iteration focuses on modifying only a single task $Ti_{bottleneck}$.

Method 3: Increment a state probability threshold P_{thresh} for removing unlikely states as described in [17], then repeat dynamic backtracking with $T_{bottleneck}$. This method is desirable only when state(s) that previously produced the backtracking failure are relatively improbable.

Each of these three methods is a possible candidate, and we are still assessing which combination of these or other techniques will provide the best backup when dynamic backtracking over $T_{bottleneck}$ fails. Currently, as referenced in Figure 4, we utilize a combination of **Method 1** and **Method 3**. With this approach, when dynamic backtracking initially fails, chronological backtracking progresses until producing a less-restrictive alternative plan. If planning progresses until even chronological backtracking fails, the planner iteratively relaxes its probability threshold P_{thresh} below which states are ignored until ultimately the planner and resource allocator can together construct a schedulable plan. A non-zero P_{thresh} value effectively degrades real-time guarantees from absolute to probabilistic. We consider this result far preferable to the alternative of CIRCA-II planning subsystem failure.

3. Resource allocation

From the perspective of 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, all guaranteed tasks are considered periodic, and each task $T_i \in T$ has worst-case computation time c_i and period P_i. The worst-case computation time includes scheduler context-switching overhead. The jth invocation of 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 algorithm to find a task schedule within a finite interval, L, equal to the least common multiple of all tasks 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.

We do not constrain how tasks are divided into modules but we assume that each module needs its resources *throughout* its execution. The selection of a proper resource allocation algorithm depends on the execution platform considered. An optimal resource allocation algorithm is described in [24] for uniprocessors, in [23] for multiprocessors, and in [21] for distributed systems. Once the task assignment is fixed, an optimal off-line scheduling algorithm such as [2] can be used to preschedule the tasks. In this paper we use [21] for task assignment and [2] for scheduling. These algorithms are used to schedule "guaranteed tasks" (those with $g_i = 1$).

Best effort tasks (with $g_i = 0$) are then fit, when possible, in the gaps of the produced schedule. The resulting overall schedule for each processor is stored in a table. If no such assignment and schedule can be found, the plan is unschedulable.

When a prescheduled plan executes, the run-time scheduler dispatches tasks in the order they appear in the table using an $O(1)$ lookup operation. Different schedules (tables) are constructed for different plans (fault conditions) and can be stored in each processor's memory, indexed by fault condition. Fault conditions may represent processor failures, communication-link failures, or other resource failures. When the condition is observed on-line, its numeric identifier is broadcast to all processors which then index into the corresponding new table. A lightweight atomic multi-cast and membership algorithm, such as [1] can be used to ensure that *all* non-failed processors (i) receive the broadcast of the current failure condition, and (ii) agree on the start time of the new schedule in bounded time despite transient communication failures. If communication is totally lost with some processor, the resulting effect is indistinguishable from a processor failure, and is treated as such. The bounded failure recovery overhead can be accounted for in off-line schedulability analysis to ensure that hard real-time constraints are met.

For completeness, in the remainder of this section we give the essential features of the resource allocation algorithm. The two main resources considered are processors and communication links. It is assumed that other resources are consumed only in conjunction with the consumption of either the processor or the communication link. For example, a module may access a common sensor while running on the processor, but presumably it cannot access the sensor if it's not running in the first place. Thus all resources other than the processor and communication link are treated as additional constraints on processor and communication link scheduling. These constraints can be precedence constraints (e.g., module A has to finish before module B can start), or mutual exclusion constraints (can't execute module A and B concurrently because they need the same additional resource and must access it one at a time). Precedence and exclusion constraints are generally called synchronization constraints.

The task assignment algorithm considers the problem of assigning m periodic tasks with known execution times, periods, and deadlines to n processors with arbitrary processing speeds such that each task meets its timing, resource, and synchronization constraints. Tasks are allowed to have allocation constraints as well, such as the constraint that two tasks must be colocated on the same processor, the constraint that two tasks must be assigned to different processors, or the constraint that a task should not be assigned to a particular processor. Each invocation of each task is composed of several modules which may represent different actions performed by the task.

The algorithm is cast as a branch-and-bound (B&B) search, by implicit enumeration, for a task allocation that minimizes maximum *task lateness*, where task lateness is defined as the difference between task completion time and task deadline. For an allocation in which all deadlines are met the maximum task lateness should be non-positive.

The algorithm considers the tasks, one at a time, for allocation to one of the processors. The root of the search is the null allocation in which no tasks have been

assigned. It is expanded by considering all possible assignments of the first task. The number of such assignments is equal to the number of processors to which the task may be assigned. Each subsequent level in the tree corresponds to assigning the next task. Thus, a vertex at level k in the tree represents an assignment of the first k tasks. Since there are m tasks in the system, vertices at level m are the leaves of the tree. They correspond to the possible solutions to the task allocation problem.

An efficient bounding function is essential for an optimal solution to be found in reasonable time. The bounding function computes a lower bound on task lateness of all allocations that include the partial allocation represented by the vertex under consideration. This lower bound for a vertex is computed in three steps:

Step 1: Compute the minimum computational load imposed on each processor by tasks already assigned in this search vertex.

Step 2: Estimate the minimum additional load to be imposed on each processor due to those tasks not yet assigned.

Step 3: Optimally schedule the combined load at each processor and compute the resulting lateness.

The exact formulae employed in these steps are described at length in [21]. Once a task assignment is reached, we use an optimal scheduling algorithm to find the minimum lateness schedule. The scheduling problem can be viewed as another B&B search. Each point in the search space represents a task schedule on processors and a message schedule on the communication links computed subject to timing and synchronization constraints. The root vertex of the search tree represents the space of all possible schedules. Branching from a vertex is a subdivision of the solution space of the parent among the set of child vertices. This is achieved by adding scheduling constraints to each child in order to limit the solution subspace represented by it.

Bounding a vertex is the estimation of a lower-bound on lateness of all the schedules in the solution subspace represented by the vertex. Bounding allows us to prune vertices whose bounds are higher (i.e., worse) than the lateness of the best schedule found so far. To enable pruning, a schedule is generated at each visited vertex out of the set of schedules in the solution subspace represented by the vertex. This is done using the earliest-deadline-first (EDF) scheduling policy. The policy generates an optimal schedule (in the sense of minimizing maximum lateness) if no exclusion constraints are present between tasks and no tasks have remote predecessors. Since the initial constraint set in the problem statement may contain both exclusion constraints and interprocessor precedence constraints, optimality of EDF is not guaranteed. Branching is therefore used to replace "unwanted" constraints by equivalent precedence constraints thus incrementally restoring the optimality of EDF. For example, assume that a pair of tasks T_1 and T_2 in the parent's schedule have a mutual exclusion constraint. To replace the mutual exclusion constraint, two children are generated, one with the added constraint T_1 *precedes* T_2 and the other with the added constraint T_2 *precedes* T_1. Since one of these precedence constraints

is necessarily satisfied in any schedule that respects the original mutual exclusion constraint, the exclusion constraint between T_1 and T_2 can be removed without affecting the space of possible solutions. Furthermore, the subsets of the solution space represented by the children are a partition of the set of schedules represented by the parent. Without proof we mention that interprocessor precedence constraints can similarly be "replaced" by modifying the deadline of the predecessor such that it inherits the lateness of its successor. The algorithm continues replacing unwanted constraints until an optimal schedule is found, i.e., all vertices have been pruned except one and no further branching is possible. Note that since branching is always possible as long as more mutual exclusion constraints or interprocessor precedence constraints are present, the latter condition implies that the EDF schedule found for the surviving vertex is optimal. More details on the algorithm are found in [3].

The combination of the task assignment algorithm and the task scheduling algorithm finds an optimal task allocation and schedule in the sense of minimizing maximum task lateness subject to individual module timing, resource, precedence, exclusion, and communication constraints. It therefore provides a powerful tool for optimal resource management in time-critical systems. We utilize this tool for prescheduling the execution of CIRCA-II plans in a distributed system such that they may execute correctly to completion subject to timing and resource constraints.

4. Planning-resource allocation interface

The primary objective of the planning-resource allocation interface is to utilize existing planning and 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" action to reconsider during replanning.

In our design, the planner transmits each plan to the interface via a set of triplets (g_i, P_i, V_i) for all tasks $T_i \in T_{total}$, where T_{total} represents all planned tasks. The guarantee flag g_i tells the interface whether the task must execute in hard real-time. The set $T_{mandatory}$ of tasks $T_i \in T_{total}$ with $g_i = 1$ exclusively dictates whether the plan is schedulable. Task period P_i is determined by the planner such that all failure transitions (TTFs) are preempted. The CIRCA-II planner assigns a priority value V_i for each task. Each V_i is given by $V_i = n_i * max(p_i)$, where n_i is the number of states in which task T_i executes and $max(p_i)$ 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.

The resource allocation analyzer receives input $(T_i \in T_{mandatory}, P_i)$ then returns a success/failure status and a utilization matrix U in which each element $U(i, q)$ is the utilization consumed by task T_i of resource q. The thread database described earlier defines the worst-case resource usage for T_i, based on the resource requirements of the modules $(M_j \in T_i)$. Elements $U(i, q)$ are computed by the resource allocation

analyzer as follows. Within each planning cycle L the total available 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 as shown in Equation 1. 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 $M_q \in T_i$, as shown in Equation 2.

$$u(k, q) = \frac{r_{k,q}C_k}{pQP_k} \tag{1}$$

$$U(i, q) = \sum_{\forall k,\, M_k \in T_i} u(k, q) \tag{2}$$

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 $\gamma_j(q)$ of each resource 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 3.

$$q_b(j) = max_q(\gamma_j(q)) = max_q\left(\sum_{i,\, i\neq j} U(i, q)\right) \tag{3}$$

The total value Sum_j remaining after eliminating T_j is the sum of V_i, as shown in Equation 4. The total value gained per unit of bottleneck-resource usage is thus $Sum_j/\gamma_j(q_b(j))$. The interface recommends for removal of the action T_j that results in the maximum value per cost ratio, as shown in Equation 5. Note that the Sum_j defined above 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.

$$Sum_j = \sum_{i,\, i\neq j} V_i \tag{4}$$

$$T_{bottleneck} = max_j\left(\frac{Sum_j}{\gamma_j(q_b(j))}\right) \tag{5}$$

The 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 is always guaranteed to find a feasible plan if one exists. The heuristic merely increases the odds of finding such a plan earlier in the search process.

Table 1. 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**

4.1. Example: selecting a bottleneck task in an unschedulable plan

We illustrate the computation of $T_{bottleneck}$ with a simple example. 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 1 describe the utilization matrix $U(i, q)$ returned from the resource allocator, 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 $U(i, q)$, resource q_3 is certainly over-utilized since the sum of individual task utilizations is greater than 1. Thus, scheduling has failed and a bottleneck task must be identified.

Table 2 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 3. 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 2 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.

5. Fault-tolerance

We establish internal fault-tolerance (e.g., to single processor failures) by using the planning–resource allocation analyzer interface module to effectively manage the preset list of faults for which the system must be tolerant. This list, F_{total}, includes the nominal "no-fault" case f_0 in which all systems work properly, and progressively

Table 2. Values used for computation of $T_{bottleneck}$

	T_1 removed	T_2 removed	T_3 removed	T_4 removed
$\gamma_1(q_1)$	0.8	0.55	0.5	0.85
$\gamma_1(q_2)$	0.55	0.5	0.7	0.65
$\gamma_1(q_3)$	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

describes more severe faults, terminating with the worst fault f_n the system can tolerate.

The CIRCA-II resource-allocation analyzer and plan execution system reference fault list F_{total}, included in the *Fault Specification List* in the *Thread Database* from Figure 3, which also contains resource type/quantity descriptions for each fault $f_i \in F_{total}$. These values are required by the *Resource Allocation Analyzer* to describe which resources are available in each fault-condition.

Figure 5 shows the interface module algorithm used to control CIRCA-II inter-module data flow. 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 cache along with indices to all faults for which that plan was "good." These faults are removed from the working fault list (placed in F_{done}), since they only require one plan. For the first (i.e., least severe) fault that over-utilized resources, a "costly" 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 alternate plan as described earlier. This procedure continues until all faults have been handled successfully by some schedulable plan.

The algorithm described above 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 cache has been filled with "good" plans for all faults, the plan indexed for nominal fault condition f_0 is selected and begins execution according to the corresponding schedule. When the system detects an internal fault, plan execution switches to the pre-scheduled plan designated to handle that fault, which has previously been stored on each plan execution processor. Thus, response to internal

1. Interface receives plan with TAPs/tasks T_{total}, including max-periods P_i, guarantee flags g_i, and priorities 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 Resource Allocation Analyzer, which returns $U(j,q)$.
   - If Resource Allocation Analyzer returns *success* status, add f_i to ''good'' 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 plan cache; reset $F_{good} = \emptyset$.
4. If $F_{done} \neq F_{total}$,
   - Find first element $f_i \in F_{total} s.t. f_i \notin F_{done}$
   - Send to planner ''costly'' task T_j identified by maximum ratio $Sum_j/\gamma_j(q_b(j))$.
   - Go to Step 1

Figure 5. Planning-resource allocation interface.

faults is prompt, and the system does not fail due to internal faults except for a fault so severe it was not incorporated into F_{total}.

6. Example: autonomous aircraft agent

We consider an example drawn from automated flight, in which rigid hard real-time response constraints require careful resource allocation and scheduling. Our plan execution system 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 two to one.

For our automated flight mission, the CIRCA-II planner is given the goals of maintaining safety while following a flight plan (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 in our example. In this section, we present a very simplified 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. 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 failure transitions, CIRCA-II selects two actions: *avoid-collision* and *maintain-trajectory*.

The decomposition of all tasks available in our flight example is shown in Tables 3 and 4. To detect a state with $OB = T$, task T_1 runs modules M_1, *scan-TCAS* (Terminal Collision and Avoidance System), to sense nearby obstacles and M_2, *monitor-*

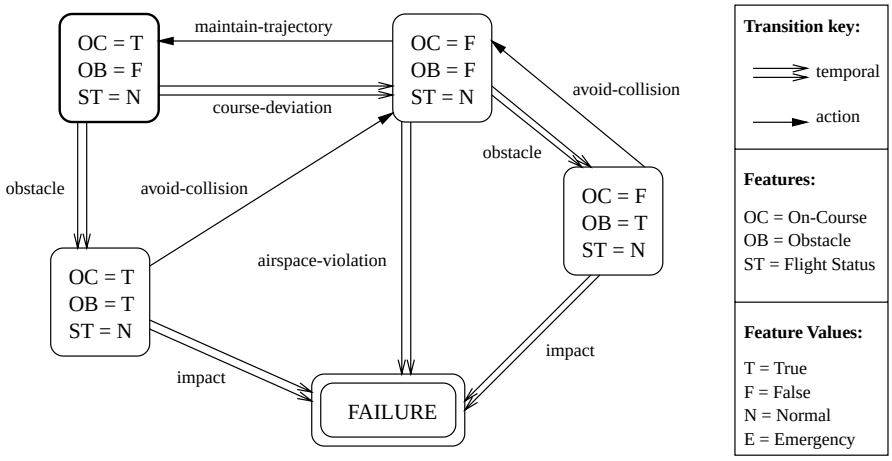

Figure 6. Nominal flight plan.

Table 3. Flight task set

$Task_i$	P_i	V_i	Modules
Avoid-collision (T_1)	6	1	M_1, M_2, M_3
Maintain-trajectory (T_2)	12	1	M_4, M_5
Declare-emergency (T_3)	6	1	M_6
Follow-radar-vectors (T_4)	12	1	M_7, M_5

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-reference*.[4]

Table 3 also includes the period (P_i) and priority (V_i) used by the CIRCA-II planner-scheduler interface. For this example, we set all task priorities equal ($V_i = 1$) because we have not yet implemented a good priority calculation algorithm in the CIRCA-II planner. In the future, V_i will be computed using a combination of state probability and temporal proximity to failure. Note that all actions are guaranteed ($g_i = 1$) since all states with planned actions have temporal transitions to failure (TTFs).

In our example, the computing system is composed of two processors, each a 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 Resource Allocation module attempts to schedule it for each fault. We consider two cases: the nominal situation where the system is fully operational (f_0) and a single processor failure (f_1). The plan is given to the resource allocation analyzer, which succeeds in computing a task assignment [21] and schedule [2] for f_0 such that all constraints are met. The resource allocation for f_0 is shown in Figure 7. Note that this allocation was obtained by considering modules M_1 through M_5 one at a time for assignment on the two processors thus generating a search tree leading to 32 possible allocations. An optimal (EDF) schedule is computed for each allocation and the vertex which results in the least maximum lateness is retained as the optimal solution. The successful plan is now added to the "good" list, F_{good}, and mode f_0 is added to the set of handled failure modes F_{done}.

Next resource allocation is attempted for the case of one processor failure, f_1. As shown in Table 5, the processor *(Proc)* utilization exceeds a value of one for f_1,

Table 4. Flight module worst-case resource usage

Module	Function	c_i on *Proc*	*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

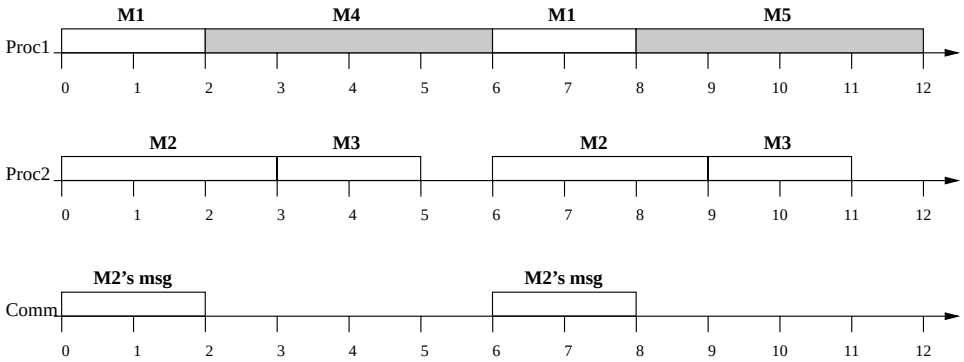

Figure 7. Nominal plan resource allocation (f_0).

thus the initial plan must be altered for f_1. The interface discovers that *Proc* is the bottleneck resource q_b for both tasks, and the interface module recommends that costly task T_1 (*avoid-collision*) be removed due to its high *Proc* utilization.

Upon dynamic backtracking to the two states where $T_{bottleneck} = T_1$ was selected, the planner determines that its only other possibility is to select task T_3 (*declare-emergency*) for these states so that ATC will re-direct traffic "obstacles" away from the aircraft. However, since task T_2 requires normal flight status, the planner also requires the addition of T_4 to handle course deviations after an emergency has been declared. The plan consisting of (T_2, T_3, T_4) also over-utilizes the one processor available with f_1. Again, the processor is the bottleneck resource q_b, and this time, the interface selects T_2 as $T_{bottleneck}$ due to its relatively intensive processor use.

The second dynamic backtracking iteration in the CIRCA-II planner yields the state diagram shown in Figure 8, with task T_3 now selected for response to either an obstacle or course-deviation when the aircraft status is normal. By declaring an emergency when fault f_1 occurs, the aircraft effectively prompts ATC to assume much of the computational responsibility. First, ATC clears the airspace so that [man-made] obstacles will no longer be a factor. Additionally, after the emergency is declared, the efficient action *follow-radar-vectors* can be utilized, in which ATC specifies the course corrections required for the aircraft to safely reach its destination.

This reduced plan (*Plan2*) shown in Figure 8 is now sent to the resource allocation analyzer, which finds the plan can easily be scheduled even with the processor failure (f_1), as computed with task utilizations shown in Table 6 and a valid task

Table 5. Utilization matrix—nominal plan

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

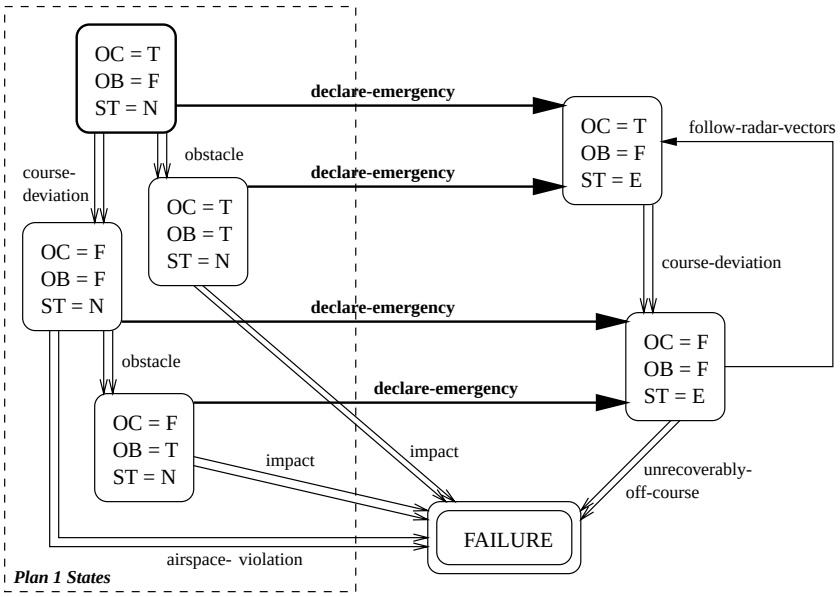

Figure 8. Reduced flight plan for failed processor (f_1).

assignment illustrated in Figure 9. With this plan, CIRCA-II is prepared to handle f_0 and f_1, so *Plan*2 is stored and planning terminates.

In this section, we have identified unschedulable plans and made them schedulable via planner-scheduler iteration. 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 the plan in the hard real-time sense.

7. Related work

The phrase "real-time" is not new to the planning community, but few architectures guarantee high quality, hard real-time response. Planning time limits have been enforced via techniques ranging from anytime [10] to design-to-time [12]. A variety of abstraction algorithms such as that from [9] allow fast, approximate planning so that a restrictive time limit will enable creation of the best plan possible within the available time. However, as domain complexity increases and available

Table 6. Utilization matrix—reduced action plan

$Task_i$	$U(i, Proc, f_1)$	$U(i, comm)$
T_3	2/12	2/12
T_4	6/12	5/12

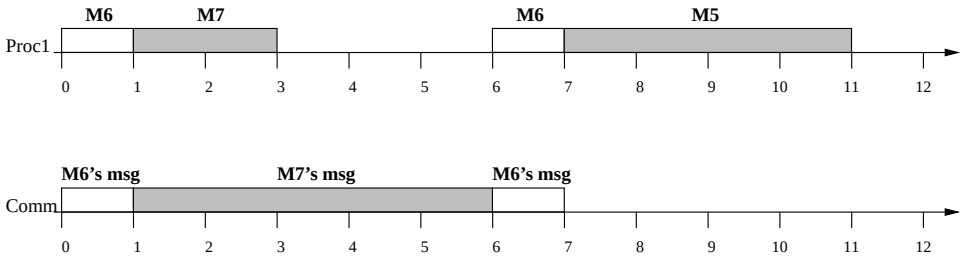

Figure 9. Resource allocation with failed processor (f_1).

planning time decreases, the quality of the planning solution may suffer to the extent that the developed plan cannot prevent system failure even though it is produced "in time." CIRCA-II does not yet provide planning timeliness guarantees, but it explicitly separates planning and plan-execution functions to minimize the need for tight planning time restrictions.

Plan-execution architectures such as PRS [15] and RAPS [11] have been developed to minimize response time by avoiding the "intractable planning" problem. Each is structured so that, if available, the appropriate reaction is discovered quickly on average, employing hierarchical techniques to limit the steps required for each search process. Although these techniques are popular in the robotics community due to their efficiency and representational power, they cannot provide absolute real-time response guarantees without strict limitations to database size (e.g., number of RAPs), which must be computed by the user in advance based on the minimum response time and worst-case search time that may be required.

The CIRCA-II plan cache is of minimal size relative to PRS and RAPS databases because the CIRCA-II cache is populated only with plans required for the particular mission at hand. However, CIRCA-II must still be able to guarantee that this size is sufficiently small to allow hard real-time retrieval times. To accomplish such guaranteed behavior, all plan cache searches are incorporated into scheduled tasks within each executing plan that, for "internal" fault-tolerance, are activated when the system must switch to a plan to handle one of the listed system faults (f_0,f_1, ...). The maximum size of the cache partition that must be searched when a fault occurs is equal to the number of user-specified faults, so the worst-case time to search for and switch to a new plan is easily predicted and automatically incorporated into the schedule for the executing plan, thereby allowing the hard real-time plan retrieval guarantees we require in CIRCA.

Architectures such as CYPRESS [22], SOAR [16], and the New Millennium Remote Architecture (NMRA) aboard the Deep Space One (DS-1) spacecraft [20] have demonstrated the ability to succeed in real-time environments. These systems combine efficiency with flexibility by using a reactive plan-execution system whenever a response is available and performing dynamic planning otherwise. However, neither explicitly reasons about task deadlines or worst-case resource utilization, so they fall under the classification of "coincidentally" real-time systems, which are appropriate only if catastrophic failure does not result when a response is too slow.

CIRCA and CIRCA-II are not the only systems to provide hard real-time plan execution guarantees. A notable exception is the conditional schedule approach demonstrated on an aircraft avionics problem in [14]. However, CIRCA-II extends beyond this and other comparable agent architectures by combining the flexibility of mission-specific planning with hard real-time guarantees that are met even in the presence of critical computational resource failures.

8. Summary and future work

We have presented an architecture that combines planning and resource allocation algorithms to produce a set of plans which execute in hard real-time on a multi-resource platform and exhibit tolerance to a user-specified set of internal systems faults. We concentrate on the interface between a state-space planner and resource allocation analyzer, which effectively provides a method for standard planning and real-time allocation algorithms to work synergistically. This interface manages a list of faults to which tolerance is required, and uses a heuristic cost function based on task priority and resource utilization to select "costly" actions for guiding the planner during backtracking should scheduling constraints be impossible to satisfy.

This work was done using a new version (CIRCA-II) of the Cooperative Intelligent Real-time Control Architecture (CIRCA), which was designed to build plans that execute with real-time CPU utilization guarantees on a uniprocessor plan execution platform. In this paper, we have described the CIRCA-II algorithms required to reason about multiple resources during plan scheduling and to develop plans that exhibit tolerance to computational resource failures. Finally, we illustrated the utility of our approach with a simple autonomous aircraft example.

We have focused on the basic mechanisms required for interfacing planning and resource allocation. However, in most real-world systems, on-line (re)planning may require that real-time bounds be placed on both planner and resource allocation modules, not just the plan-execution system. As proposed in [5], CIRCA-II may utilize its plan cache to "buy time" so that real-time constraints on planning and resource allocation are as relaxed as possible, but still assumptions of "indefinite replanning time" may be inappropriate. By introducing the new iterative interface module, we have made resource-bounded planning even more difficult to achieve. We hope to address this formidable problem by using efficient resource allocation algorithms with bounded execution constraints, incorporating anytime [10] techniques to planning, and limiting replanning iterations to accommodate a subset of the faults from F_{total}, if required.

Experimental validation of our algorithms is in progress with the University of Michigan Uninhabited Aerial Vehicle (UAV) Project, described in [8]. Our UAV is a radio-controlled (R/C) aircraft with onboard and ground-based processing systems. It is sufficiently instrumented to ultimately allow autonomous operation via low-level control software that we are connecting to CIRCA-II for higher-level mission planning tasks. We plan to incorporate CIRCA-II along with adaptive model identification algorithms to study aircraft response to a variety of situations including both environmentally-induced emergencies (e.g., engine failure, airframe icing)

and internal faults (e.g., sensor, communication, or processor failures). We are confident that testing the UAV in these situations will reveal means of improving both CIRCA-II and adaptive control algorithms, and will clearly demonstrate the utility of a system that has the flexibility to build plans from a knowledge base but can still guarantee hard real-time response characteristics and fault tolerance.

Acknowledgments

This work was partially supported by the NSF under Grant IRI-9209031 and by the ONR under Grant N00014-94-1-0229.

Notes

1. To allow use of the well-developed set of real-time scheduling algorithms for *periodic* tasks, we have converted CIRCA's task separation constraints into task periods P_i. As described in [18], task periods must be artificially set below their optimal values to guarantee meeting the former task separation constraints. We are working to better represent task periods in future CIRCA-II work.
2. In fact, we are still in the process of implementing the full multi-resource run-time scheduler within the Real-time Plan Executor. Thus, in this paper we present multi-resource plans, not post-execution results.
3. Backtracking *not* induced by plan scheduling failures utilizes different algorithms described in [4, 19]
4. 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.

References

1. T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: lightweight multicast for real-time process groups," in *IEEE Real-Time Technol. Applicat. Symp.*, Boston, MA, 1996.
2. T. F. Abdelzaher, and K. G. Shin, "Optimal combined task and message scheduling in distributed real-time systems," in *IEEE Real-Time Syst. Symp.*, Pisa, Italy, 1995.
3. T. F. Abdelzaher, and K. G. Shin, "Combined task and message scheduling in distributed real-time systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 11, 1999, pp. 1179–1191.
4. E. M. Atkins, "Plan generation and hard real-time execution with application to safe, autonomous flight," Ph.D. dissertation, University of Michigan, 1999.
5. E. M. Atkins, E. H. Durfee, and K. G. Shin, "Plan development using local probabilistic models," in *Proc. Twelfth Conf. Uncertainty Artif. Intell.*, 1996, pp. 49–56.
6. E. M. Atkins, E. H. Durfee, and K. G. Shin, "Buying time for resource-bounded planning," in *AAAI-97 Workshop: Building Resource-Bounded Reasoning Systems Technical Report*, 1997, pp. 7–11.
7. E. M. Atkins, E. H. Durfee, and Kang G. Shin, "Detecting and reacting to unplanned-for states," in *Proc. Fourteenth Nat. Conf. Artif. Intell.*, 1997, pp. 571–576.
8. 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 *Proc. Am. Contr. Conf.*, vol. 2, 1998, pp. 689–693.
9. C. Boutilier, and R. Dearden, "Using abstractions for decision-theoretic planning with time constraints," in *Proc. Twelfth Nat. Conf. Artif. Intell.*, 1994, pp. 1016–1022.
10. T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson, "Planning with deadlines in stochastic domains, in *Proc. Eleventh Nat. Conf. Artif. Intell.*, 1993, pp. 574–579.

11. R. J. Firby, "An investigation into reactive planning in complex domains," in *Proc. Natl. Conf. Artif. Intell.*, 1987, pp. 202–206.
12. A. J. Garvey and V. R. Lesser, "Design-to-time real-time scheduling," *IEEE Trans. Syst. Man Cybernet*. vol. 23, no. 6, pp. 1491–1502, 1993.
13. M. L. Ginsberg, "Dynamic backtracking," *J. Artif. Intell. Res.* vol. 1, pp. 25–46, 1993.
14. L. Greenwald, and T. Dean, "Solving time-critical decision-making problems with predictable computational demands," in *Proc. Second Int. Conf. AI Planning Syst.*, 1994.
15. F. F. Ingrand, and M. P. Georgeff, "Managing deliberation and reasoning in real-time AI systems," in *Proc. Workshop on Innovative Approaches to Planning, Scheduling, Control*, 1990, pp. 284–291.
16. J. E. Laird, A. Newell, and P. S. Rosenbloom, "SOAR: an architecture for general intelligence," *Artif. Intell.*, vol. 33, pp. 1–64, 1987.
17. C. B. McVey, E. M. Atkins, E. H. Durfee, and K. G. Shin, "Development of iterative real-time scheduler to planner feedback," in *Proc. Fourteenth Int. Joint Conf. Artif. Intell.*, 1997, pp. 1267–1272.
18. 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, 1994.
19. D. J. Musliner, E. H. Durfee, and K. G. Shin, "World modeling for the dynamic construction of real-time control plans," *Artif. Intell.* vol. 74, pp. 83–127, 1995.
20. B. Pell, E. Gat, R. Keesing, N. Muscettola, and Ben Smith, "Plan execution for autonomous spacecraft," in *AAAI-96 Fall Symp. Plan Execution: Problems and Issues Technical Report*, 1996, pp. 109–116.
21. D.-T. Peng, and K. G. Shin, and T. F. Abdelzaher, "Assignment and scheduling of communicating periodic tasks in distributed real-time systems," *IEEE Trans. Parallel Distrib. Syst.* vol. 8, no. 12, 1997.
22. D. E. Wilkins, K. L. Myers, and J. D. Lowrance, "Planning and reacting in uncertain and dynamic environments," *J. Exp. Theor. AI* vol. 7, no. 1, pp. 197–227, 1995.
23. J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Eng.* vol. 19, no. 2, pp. 139–154, 1993.
24. J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Eng.* SE-vol. 16, no. 3, pp. 360–369, 1990.