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

Ella M. Atkins Tarek F. Abdelzaher Kang G. Shin Edmund H. Durfee
Department of Electrical Engineering and Computer Science
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
{marbles, zaher, kgshin, durfee}@umich.edu

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 allocator schedules these plans off-line to analyze execution platform 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.

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.

The real-time research community focuses its efforts on resource allocation and scheduling algorithms to provide hard real-time execution guarantees. Violating timing constraints of such tasks may be catastrophic. 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 dependency-directed 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 envisioned potential faults in 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) [1]. 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 [2]. 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 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 and 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 present a simple example of an autonomous aircraft agent that must be tolerant to a single-processor failure during plan execution to illustrate the utility of our algorithms, then describe related work in planning and plan-execution systems. We conclude with a summary and discussion of work required to further test our system and bridge the gap between the planning and hard real-time research communities.

2 Planning for Hard Real-time Execution

We selected the Cooperative Intelligent Real-time Control Architecture (CIRCA) 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 describe CIRCA as it has appeared in previous work, then outline improvements that enable tolerance to "internal faults" (i.e., traditional computational system faults as opposed to environmental occurrences that we label "external faults"). We also discuss our addition of scheduling/allocation algorithms to allow multiple resource instances and classes (e.g., multiple processors, multiple communication channels) for CIRCA plan execution.

Figure 1 shows the CIRCA architecture originally developed by Musliner et al [1]. 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 [3] which promotes a best-first state-space search as well as limiting search size via removal of "highly improbable" states [4]. 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,

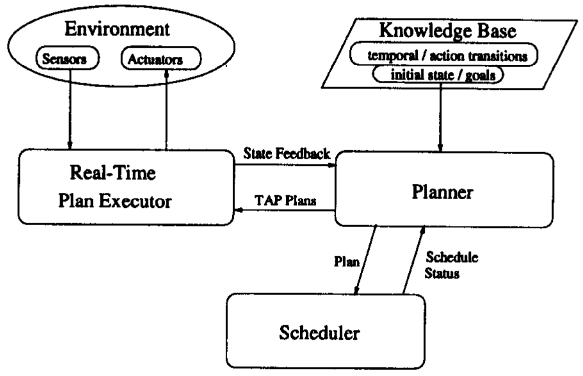

Figure 1: Original CIRCA.

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*** and ***best-effort***. Guaranteed tasks have hard real-time execution constraints (i.e., max-periods) 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.

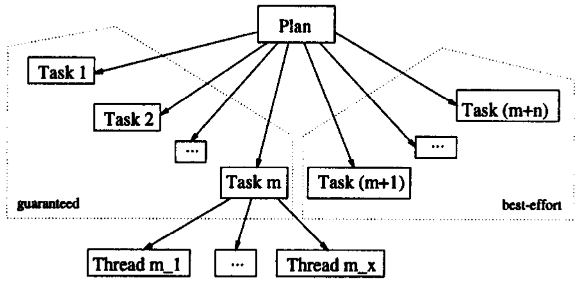

Figure 2: CIRCA plan composition.

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 processors nor other required resources, such as communication channels, could be individually considered during scheduling. As a result, not only was the plan-execution platform inflexible, but there was no possibility for "internal" fault-tolerance since multiple copies of computational resources could not exist.

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 CIRCA architectural modifications required to gain these new capabilities. The knowledge base and planner are very similar to those from previous versions of CIRCA. Plans are created and action timing constraints are computed as before in [1] and [3]. 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 max-period of T_i required to preempt TTFs when $g_i = 1$. Finally, V_i is the "priority" value of task T_i and is described below in Section 4.

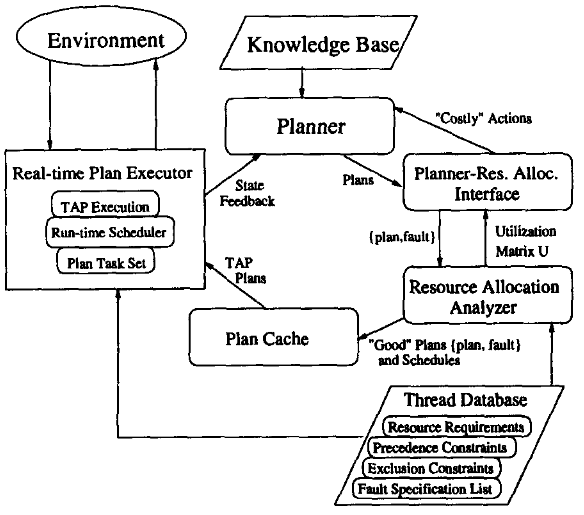

Figure 3: Modified CIRCA.

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 requested by the plan executor.

When a plan is unschedulable, the interface computes the most costly action, which is fed back to the planner. The planner attempts to remove or replace this action via dependency-directed backtracking to each state in which this action was chosen. Ideally, this action can be modified (e.g., max-period P_i increased) or replaced, in which case the planner develops a new plan. If this action cannot be altered, the planner initially ignores the interface's recommendation and modifies the next action encountered during chronological backtracking, the default in CIRCA. Minimally-modified plans developed in this manner are assessed for schedulability, with planner–resource allocation iteration continuing until a schedulable plan is found. If the planner exhaustively searches through all combinations of actions that pre-empt TTFs without finding a schedulable plan, the planner iteratively relaxes its probability threshold below which failure transitions are ignored until a schedulable plan is found, effectively degrading real-time guarantees from absolute to probabilistic-by-necessity. In [2], the latter technique of incrementing probability threshold was the sole method employed for improving plan schedulability. By recommending a costly action for removal, we help the planner replace costly actions with different actions while maintaining absolute real-time guarantees whenever possible.

The plan cache was added specifically for storing plans to handle system faults. As internal systems fail, the planner may compromise on overall plan quality to allow safety guarantees, even if they are probabilistic. We store a set of plans, indexed by fault condition, so that the system can respond to faults during plan execution without replanning.[1] CIRCA's real-time plan-execution system has increased flexibility due to the capability to include multiple resources (e.g., processors, communication channels). As is a standard practice in the real-time community, resources are reserved for run-time scheduler execution, which allows flexible utilization of available resources to handle faults and incrementally improve "best-effort" task execution when sufficient resources are available.

3 Resource Allocation

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, 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.

The selection of a proper resource allocation algorithm depends on the execution platform considered. An optimal resource allocation algorithm is described in [6] for uniprocessors, in [7] for multiprocessors, and in [8] for distributed systems. Once the task assignment is fixed, an optimal off-line scheduling algorithm such as [9] can be used to preschedule the tasks. In this paper we use [8] for task assignment and [9] 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.

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 multicast and membership algorithm, such as [10] can be used to ensure that *all* non-failed processors (i) receive the broadcast of the current failure condition, and (ii) agree on the

[1] A CIRCA plan cache has been proposed in [5] for achieving timely reaction to "unhandled" states. The plan cache in this paper contains multiple plans only for fault-tolerance.

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.

4 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" 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 max-period P_i is determined by the planner such that all failure transitions (TTFs) are preempted. We use a heuristic in CIRCA's planner to compute 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 $(T_i \in T_{mandatory}, P_i)$ and 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 (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 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. Thus, $u(k,q) = r_{k,q}C_k/pQP_k$. 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 $M_q \in T_i$.

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 of each resource q by adding $U(i,q)$ for all i, where $i \neq j$. Let $\gamma_j(q) = \sum_{i,i\neq j} U(i,q)$. The bottleneck resource $q_b(j)$ is the one for which $\gamma_j(q)$ is maximum. The total value remaining after eliminating T_j is the sum of V_i for all i, $Sum_j = \sum_{i,i\neq j} V_i$. 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, $Sum_j/\gamma_j(q_b(j))$. 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.

The heuristic is used to suggest which part of the planner's search space to expand next, but does not 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. The problem, however, remains NP-complete.

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

The CIRCA 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 4 shows the interface module algorithm used to control CIRCA's 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 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

```
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 4: Planning–resource allocation interface.

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 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 5. 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 selects two actions: *avoid-collision* and *maintain-trajectory*.

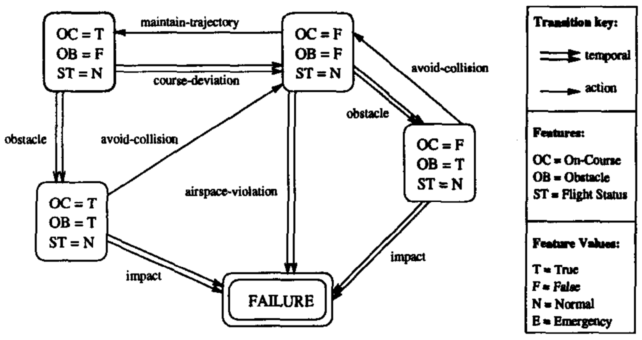

Figure 5: Nominal Flight Plan.

The decomposition of all tasks available in our flight example is shown in Tables 1 and 2. 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-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*.[2]

$Task_i$	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

Table 1: Flight Task Set.

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

Table 2: Flight Module Worst-Case Resource Usage.

Table 1 also includes the period (P_i) and priority (V_i) used by the CIRCA's 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 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 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 [8] and schedule [9] for f_0 such that all constraints are met. The resource allocation for f_0 is shown in Figure 6. 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}.

As shown in Table 3, the processor (*Proc*) utilization exceeds a value of one for f_1, thus the initial plan must be altered for f_1. Since *Proc* is the bottleneck, the interface module recommends that the planner remove T_1 (*avoid-collision*) due to its high *Proc* utilization.

Backtracking during replanning yields the state diagram shown in Figure 7, with the new task *declare-emergency*(T_3)

[2]CIRCA 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.

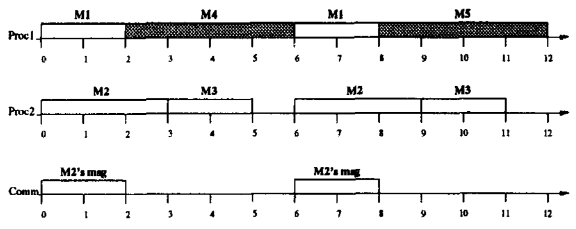

Figure 6: Nominal Plan Resource Allocation (f_0).

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

Table 3: Utilization Matrix – Nominal Plan.

selected.[3] Once the emergency is declared, ATC effectively takes much of the computational responsibility from the aircraft, clearing the airspace so that obstacles will no longer be a factor. 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.

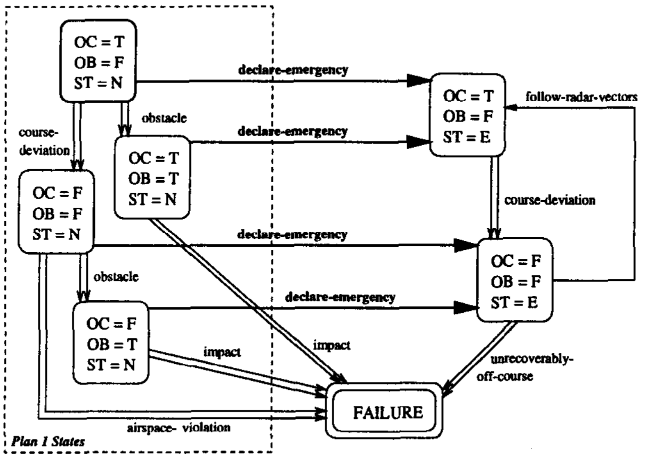

Figure 7: Reduced Flight Plan for Failed Processor (f_1).

This reduced plan (*Plan2*) 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 4 and a valid task assignment illustrated in Figure 8. With this plan, we can now handle both f_0 and f_1, so *Plan2* is stored and planning terminates.

In this section, 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 the plan in the hard real-time sense.

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

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

Table 4: Utilization Matrix – Reduced Action Plan.

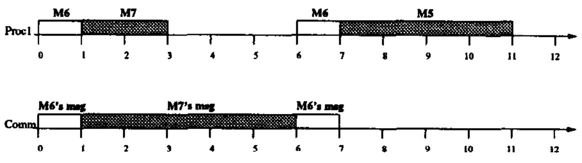

Figure 8: Resource Allocation with Failed Processor (f_1).

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 [11] to design-to-time [12]. A variety of abstraction algorithms such as that from [13] 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 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 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 [14] and RAPS [15] 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 plan cache is of minimal size relative to PRS and RAPS databases because the CIRCA cache is populated only with plans required for the particular mission at hand. However, CIRCA 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 [16] and SOAR [17] 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.

Other systems, such as the AI system to control the Deep Space One (DS-1) spacecraft [18] and the conditional schedules demonstrated on an aircraft avionics problem in [19], do provide timeliness guarantees during plan execution, but incorporate neither "internal" fault-tolerance nor the flexibility to perform allocation and scheduling in a variety of multi-resource environments.

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 the Cooperative Intelligent Real-time Control Architecture (CIRCA), which was originally designed to build plans that execute with real-time CPU utilization guarantees on a uniprocessor plan execution platform. We described the augmentation of CIRCA to accommodate multiple plan execution resources and to exhibit fault-tolerance during plan execution, and provided an example illustrating the utility of our approach.

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 may be augmented to use the 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 [11] techniques to planning, and limiting replanning iterations to accommodate a subset of the faults from F_{total}, if required.

Experimental validation of our CIRCA algorithms is in progress with the University of Michigan Uninhabited Aerial Vehicle (UAV) Project, described in [20]. Our UAV is a radio-controlled (R/C) aircraft with onboard and ground-based processing systems. It is fully-instrumented with sensors including GPS, IMU (inertial measurement unit), air data system, tachometer, and control surface displacement sensors that will allow autonomous operation via low-level control software that we are connecting to CIRCA for higher-level mission planning tasks. We plan to incorporate CIRCA 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 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.

9 Acknowledgements

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

References

[1] D. J. Musliner, E. H. Durfee, and K. G. Shin, "World modeling for the dynamic construction of real-time control plans," *Artificial Intelligence*, vol. 74, pp. 83–127, 1995.

[2] C. B. McVey, E. M. Atkins, E. H. Durfee, and K. G. Shin, "Development of iterative real-time scheduler to planner feedback," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1267–1272, August 1997.

[3] E. M. Atkins, E. H. Durfee, and K. G. Shin, "Plan development using local probabilistic models," in *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pp. 49–56, July 1996.

[4] E. M. Atkins, E. H. Durfee, and K. G. Shin, "Detecting and reacting to unplanned-for states," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 571–576, July 1997.

[5] 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*, pp. 7–11, July 1997.

[6] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Engineering*, vol. SE-16, no. 3, pp. 360–369, March 1990.

[7] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 139–154, February 1993.

[8] D.-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*, vol. 8, no. 12, , December 1997.

[9] T. F. Abdelzaher and K. G. Shin, "Optimal combined task and message scheduling in distributed real-time systems," in *IEEE Real-Time Systems Symposium*, Piza, Italy, December 1995.

[10] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight multicast for real-time proces groups," in *IEEE Real-Time Technology and Applications Symposium*, Boston, MA, June 1996.

[11] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson, "Planning with deadlines in stochastic domains," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 574–579, July 1993.

[12] A. J. Garvey and V. R. Lesser, "Design-to-time real-time scheduling," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 6, pp. 1491–1502, 1993.

[13] C. Boutilier and R. Dearden, "Using abstractions for decision-theoretic planning with time constraints," in *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1016–1022, July 1994.

[14] F. F. Ingrand and M. P. Georgeff, "Managing deliberation and reasoning in real-time AI systems," in *Proc. of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pp. 284–291, November 1990.

[15] R. J. Firby, "An investigation into reactive planning in complex domains," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 202–206, August 1987.

[16] D. E. Wilkins, K. L. Myers, and J. D. Lowrance, "Planning and reacting in uncertain and dynamic environments," *Journal of Experimental and Theoretical AI*, vol. 7, no. 1, pp. 197–227, 1995.

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

[18] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith, "Plan execution for autonomous spacecraft," in *AAAI-96 Fall Symposium on Plan Execution: Problems and Issues Tech. Report*, pp. 109–116, November 1996.

[19] 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.

[20] 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*, volume 2, pp. 689–693, June 1998.