A Decision-Theoretic Planner with Dynamic Component Reconfiguration for Distributed Real-Time Applications

John S. Kinnebrew, Ankit Gupta, Nishanth Shankaran, Gautam Biswas, Douglas C. Schmidt

Department of Electrical Engineering and Computer Science & ISIS,
Vanderbilt University, Nashville, TN 37203, USA
john.s.kinnebrew@vanderbilt.edu

Abstract

Distributed real-time embedded (DRE) systems perform sequences of coordination and heterogeneous data manipulation tasks in dynamic environments to meet specified goals. Autonomous operation of DRE systems can benefit from the integrated operation of (1) a decision-theoretic Spreading Activation Partial Order Planner *(SA-POP) that combines task planning and scheduling in uncertain environments with (2) a* Resource Allocation and Control Engine *(RACE) middleware framework that integrates multiple resource management algorithms for (re)deploying and (re)configuring task sequence components in these systems. This paper demonstrates the effectiveness of SA-POP and RACE in managing and executing mission goals for a multi-satellite application. Our results show that combining planning, scheduling and resource constraints dynamically is the key to implementing autonomy in DRE systems.*

1. Introduction

Distributed real-time embedded (DRE) systems, such as multi-satellite and multi-robot formations, often perform sequences of heterogeneous data collection, manipulation and coordination tasks to meet specified goals. For example, weather prediction requires multiple satellites equipped with different sensors flying coordinated missions to collect and analyze large quantities of atmospheric and earth surface data. The data collection, analysis, and earth transmission tasks may change during operation as previously collected data indicates other factors or regions of interest, and overall goals and priorities have to be modified with changing weather patterns or uncertainties attributed to changing resource availability. Moreover, limited bandwidth and communication lag necessitate autonomous (re)planning on-board the satellites to effectively achieve goals under rapidly evolving conditions.

Presently task sequence implementations in DRE systems use *component middleware* [5], which automates remoting, lifecycle management, system resource management, deployment, and configuration. In large DRE systems, the sheer number of component sequences poses a combinatorial problem of mapping components to computing nodes [8]. The dynamic nature of the operations requires runtime management and modification of deployments [4]. More effective solutions providing greater autonomy must include planning and replanning capabilities, and dynamic monitoring and runtime management to ensure chosen task sequences keep in sync with changing mission goals and resource availability.

For example, the NASA Earth Science Enterprise's Magnetospheric Multi-Scale (MMS) [3] mission uses five coordinated satellites as a solar-terrestrial probe. Each of the satellites uses six sensors to collect electromagnetic and particle data in the earth's magnetosphere. The mission operates in three data modes: slow, fast, and burst. These data modes imply different goals, orbits, and data priorities. Each satellite has to determine its task sequences to achieve prescribed goals based on the current environmental and system conditions, and revise its task sequences when conditions change.

To achieve this degree of autonomy the system uses an automated planner that can handle changing goal prescriptions specified by mission scientists, and autonomous mode changes driven by satellite position and the results of data analysis. The planner handles multiple interacting goals for coordinating the trajectory and orientation of satellites, sensor selection and data collection for individual satellites, and data integration and compression to create telemetry streams that are transmitted to the earth stations. In addition, the runtime computational architecture includes dynamic monitoring schemes and resource reallocation schemes to accommodate changing resource use during operation. Replanning algorithms may be invoked to accommodate changing goals and operating conditions.

To support DRE systems, we have developed a novel, computationally efficient algorithm called the *Spreading Activation Partial Order Planner* (SA-POP) for dynamic (re)planning under uncertainty. SA-POP overcomes scaling limitations of earlier AI algorithms for combined planning and resource allocation/scheduling [19]. This paper describes our approach for combining SA-POP with a *Resource Allocation and Control Engine* (RACE), a reusable component middleware framework that separates resource allocation and control algorithms from the underlying middleware deployment, configuration, and control mechanisms to enforce quality of service (QoS) requirements.

2. DRE system architecture

Figure 1 shows the system architecture for autonomous operations of complex DRE applications using SA-POP and RACE. The SA-POP planner starts with specified mission goals and generates *operational strings* that represent appropriate task sequences of high expected utility. Each goal specification maps onto one operational string,

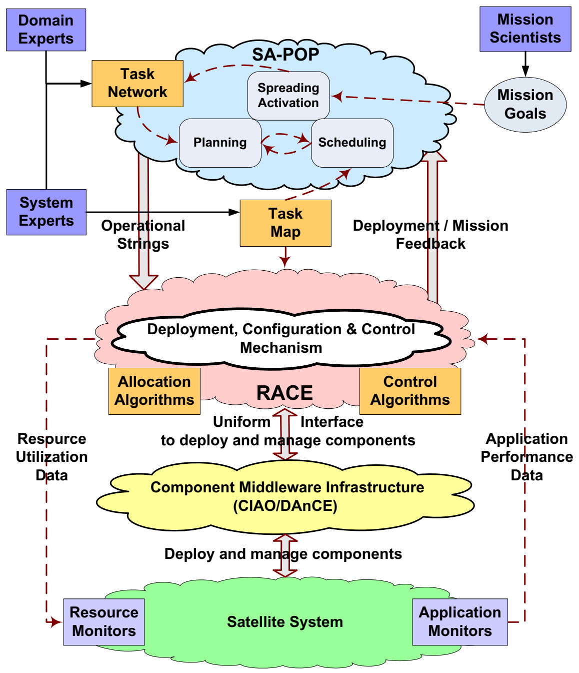

Figure 1. SA-POP and RACE architecture

which includes the control (ordering) dependencies, the data (producer/consumer) dependencies, and required start and end times for tasks, if any. The operational strings also contain suggested implementations for each task. RACE performs the initial deployment by mapping task implementations onto computational resources, and then monitors and manages runtime resource allocation to enforce QoS requirements.

DRE systems in the domains of shipboard computing [15], avionics mission computing [17], and intelligence, surveillance and reconnaissance [16] often represent applications as groups of domain-related tasks modeled as operational strings. These strings are implemented by executable software components using component technologies, such as the OMG's Lightweight CORBA Component Model (CCM) [12] and Web Services. In our architecture, *components* are implementation units that contain parameterized executable code with resource consumption profiles (such as expected CPU and memory usage) and specified QoS requirements (*e.g.*, maximum latency and minimum throughput.

For the MMS application to achieve a given set of *goals* (*e.g.*, study the physics of plasma reconnection and charged particle acceleration), SA-POP uses a spreading activation mechanism [1] to generates expected utility values for individual tasks that can contribute to achieving specified goals. Guided by these expected utility values, SA-POP's planning and scheduling algorithms generate partial order *task sequences* from which the operational strings are derived. The individual tasks in a sequence are then mapped to available executable software components, *e.g.* a data compression task may be mapped into an appropriate compression algorithm.

For a task to execute successfully, SA-POP must know its preconditions, its input/output data stream, and other effects it produces. Uncertainty in task execution and the output generated is captured as conditional probabilities associated with the preconditions and effects of a task. The input/output definitions, preconditions/effects, and related conditional probabilities define the *functional signature* of the task. Different parameterizations of a given component may produce different functional signatures. Conversely, different components that have the same functional signature may vary in time to completion, resource usage, and QoS parameters.

We define a *task* as one or more parameterized components with a single functional signature. The functional signature of tasks and their dependencies are captured in a *task network*, which is a directed graph that represents both tasks and conditions (preconditions, data input, effects, and data output) with the links encoding the requisite probability information. The task network can be constructed by domain experts using domain-specific modeling tools (*e.g.*, the Generic Modeling Environment (GME) [6]). With the task network and user-specified utility values for goal conditions/data, a spreading activation mechanism, described in the next section, computes expected utility values for each task.

To ensure applications do not violate resource constraints, SA-POP also requires *resource signatures*, i.e., resource consumption and execution time for each task implementation. A task may have multiple parameterized component implementations with different resource signatures. SA-POP uses a *task map* to associate tasks with their parameterized components and corresponding resource signatures.

Operational strings produced by SA-POP are input to RACE, which uses reusable algorithms to (1) deploy the initial mapping of components to nodes and (2) monitor system and application resource usage [14] to manage system performance. RACE allocates resources to application components based on their resource requirements and QoS characteristics. Since component resource use and end-to-end QoS for operational strings are sensitive to runtime changes and changes in system performance, *e.g.*, due to changes in resource availability and transient overload, RACE can also redeploy and/or reconfigure application components using the implementation options available in the task map to ensure the desired end-to-end QoS requirements of operational strings are not violated.

3. SA-POP

Autonomous operations in DRE systems require two components that operate in concert: (1) a planning and scheduling system that responds to changes in goal specifications, environmental conditions, and changes brought about by the interpretation of collected data, and (2) a resource allocation engine that can monitor resource usage and QoS specifications to ensure that the plan execution meets the desired goal specifications. This section describes the SA-POP planner and scheduler, which include (1) a decision-theoretic spreading activation mechanism to identify task sequences that maximize an expected utility measure for given a set of goals, and (2) an operational string generation mechanism that uses the task expected utilities and their implementation resource signatures to ensure that the generated operational string have high expected utility and meet resource, time, and other QoS constraints.

3.1. Spreading activation networks

A spreading activation task network, shown in Figure 2, captures the links between task sequences and goal conditions [1]. The network contains condition nodes (ovals) and task nodes (rectangles) with directed links that indicate the pre- and post-conditions for executing individual tasks. Condition nodes are represented as Boolean variables with associated probabilities that define the maximum likelihood

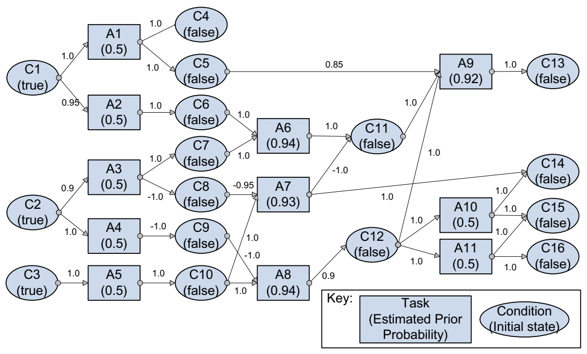

Figure 2. An example task network

of that node achieving true/false values. Environmental/-system conditions (*e.g.*, a particular sensor is active) and generated data (*e.g.*, a data stream from a sensor) are represented as condition nodes. The data condition nodes represent the availability (true) or non-availability (false) of the corresponding data.

The weight, w_{ij}, of the link from a condition node, c_i, to a task node, t_j, defines the likelihood that t_j succeeds in given c_i, *i.e.*

$$w_{ij} = \frac{P(t_j^s|c_i = true) - P(t_j^s|c_i = false)}{P(t_j^s|c_i = true) + P(t_j^s|c_i = false)}, \qquad (1)$$

where t_j^s indicates that task t_j is successful. This encoding supports *hard constraints* (weight = 1 (-1)), *i.e.*, where the condition must be true (false) for the task to succeed, and *soft constraints* (weight < 1 (> -1)), *i.e.*, where the true (false) value of the condition increases the probability of task success. Soft constraints can be used to model inferred conditions in uncertain environments, where an actual precondition can not be sensed directly but is probabilistically related to other conditions that can be sensed. For example, an imperfect (noisy) sensor for detecting an environmental condition necessary to the success of a task can be modeled using a soft constraint. The weight, w_{jk}, of the link from a task node, t_j, to a condition node, c_k, defines the probability that c_k will be true/false after t_j executes, *i.e.*:

$$w_{jk} = \begin{cases} P(c_k = true|t_j^x) & \text{if } t_j \text{ sets } c_k = true \\ -P(c_k = false|t_j^x) & \text{if } t_j \text{ sets } c_k = false, \end{cases} \qquad (2)$$

where t_j^x indicates that task t_j is executed.

The likely contribution of a task toward a desired goal is computed as an expected utility (EU), *i.e.*, the product of the task's utility toward meeting the goal requirements and its likelihood of success. Probability values are propagated forward through the network from preconditions through tasks to effects. Utility values are propagated backward through the network from effects through tasks to preconditions, which allows preconditions of potentially useful tasks to accumulate utility and makes them useful subgoals toward meeting the specified goal requirements.

Two experiments using the task network in Figure 2 illustrate the results of spreading activation with different goals. In experiment 1, condition nodes C14 and C15 each have a goal utility of 100, whereas in experiment 2, C15 is replaced with C16, again with a goal utility of 100. Table 1 summarizes the resulting expected utility and probability values for all task and condition nodes in the network. Note

Nodes	EU Exp 1	EU Exp 2	Prob Exp 1	Prob Exp 2
A1	0	0	1.000	1.000
A2	0	0	0.975	0.975
A3	0	0	0.950	0.950
A4	0	0	1.000	1.000
A5	181	181	1.000	1.000
A6	0	0	0.926	0.926
A7	100	100	0.990	0.990
A8	162	162	1.000	1.000
A9	0	0	0.830	0.830
A10	180	90	0.900	0.900
A11	90	90	0.900	0.900
C1	0	0	1.000	1.000
C2	0	0	1.000	1.000
C3	181	181	1.000	1.000
C4	0	0	1.000	1.000
C5	0	0	1.000	1.000
C6	0	0	0.975	0.975
C7	0	0	0.950	0.950
C8	-95	-95	0	0
C9	-162	-162	0	0
C10	181	181	1.000	1.000
C11	0	0	0.926	0.926
C12	180	180	0.900	0.900
C13	0	0	0.830	0.830
C14	99	99	0.990	0.990
C15	90	0	0.900	0.900
C16	0	90	0.900	0.900

Table 1. Spreading activation results

that negative EU values indicate expected utility for a condition being false instead of true, and the probability values listed are probabilities of success for tasks and probabilities of a true value for conditions. Figures 3 and 4 illustrate the different plans that result based on expected utilities in the two experiments.

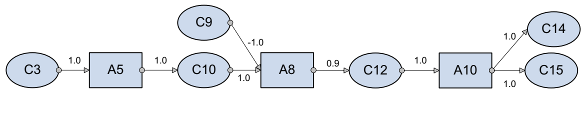

Figure 3. Resultant plan in experiment 1

Since C15 is not a goal in experiment 2, A10 only accumulates utility from C14. As a result, A7 has a higher expected utility than A10 due to its higher probability of success. Therefore, it is chosen to achieve C14 instead of A10. Moreover, A11 is also necessary to achieve the goal of C16 in experiment 2, while A5 and A8 are common to

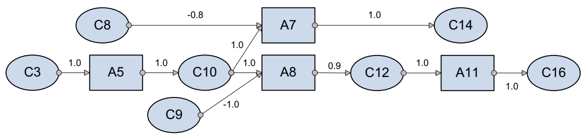

Figure 4. Resultant plan in experiment 2

both plans as they achieve subgoals necessary for following tasks.

3.2. Operational string generation

For the NASA MMS and similar DRE systems, the fewer the constraints imposed by an operational string, the easier it is to make initial deployment decisions and manage resources at runtime. To facilitate these activities, we adopt a modified *Partial Order Causal Link* (POCL) design [18] to generate operational strings. The least commitment strategies typical of partial order planning allow SA-POP to impose relatively few constraints compared to other popular planning techniques, such as state space search and constraint satisfaction based planners. Recent research [9] also indicates that in many cases the performance of partial order planning can be brought up to par with these other approaches by applying appropriate heuristics.

SA-POP leverages information from the partial order planning process when applying resource constraints and finding resource violations. In DRE systems, such as the MMS scenario, many data manipulation tasks operate over long time windows with a required start time, but the end time is dynamically determined by ongoing analysis of the data, which limits the effectiveness of many popular scheduling approaches such as timetabling [13], edge-finding [2], and classical energetic reasoning [7].

Rather than primarily relying on start/end time window manipulation, SA-POP leverages the ordering constraints common to partial order plans. These constraints are used to create *precedence graphs* [7], which partition all other tasks into sets based on their ordering with respect to a particular task under consideration. SA-POP then applies a modified version of Laborie's energy precedence and balancing constraint propagation techniques [7] to specify four different kinds of "links," which specify an ordering between two task instances. A *Causal Link* indicates one task must execute before the other based on a system/environmental condition. A *Data Link* indicates both tasks must execute simultaneously because they both operate on the same data stream. A *Threat Link* indicates one task must execute before the other. A *Scheduling Link* indicates one task must execute before the other. The links are imposed during scheduling to prevent potential resource violations, and are valid within, as well as across, operational strings. We also define one additional type of constraint on task instances in an operational string, a *Time Constraint*. This constraint specifies a required start-by or end-by time, and it is specified in the goal input for a required condition.

SA-POP also maintains additional time and ordering information, such as the *Time Window*, which consists of an earliest time and latest time for each task instance. A *Ranking*(a, b) is a comparison between task instances a and b, which describes the order in which they will be executed given the current knowledge of the plan. There are four rankings used by SA-POP: *Before* (a will complete its execution before b begins executing), *After* (reciprocal of the Before relation), *Simultaneous*(both a and b will start and end their executions strictly at the same times), and *Unranked* (a and b overlap or potentially overlap).

The rankings between all pairs of tasks are maintained in a precedence graph [7]. The precedence graph maintained by SA-POP differs from Laborie's definition primarily in that it is defined between pairs of task instances rather than *events*, which are the individual start and end times of task instances. This simplification allows more efficient scheduling calculations for *discrete resources*, *e.g.*, memory which is used during a task execution and then freed. SA-POP currently does not deal with *reservoir resources*, which are resources, such as battery power, that can be arbitrarily produced or consumed.

SA-POP generates operational strings using mutually recursive planning and scheduling algorithms with backtracking. Each step in the generation of an operational string involves the four recursive algorithms described below:

Algorithm: *Plan*. SA-POP begins with the mission goals as the set of open conditions. Since data manipulation tasks are resource intensive and execute concurrently with other tasks on the same data stream, SA-POP gives priority to data flow conditions, which enables early detection of irresolvable resource violations in a nascent plan, thereby pruning the search space. In choosing a task to satisfy the current open condition, SA-POP prefers tasks with higher expected utility values, weighted by their probability of achieving the condition. There is also a threshold on this probability value, and those tasks falling below the threshold are ranked strictly by probability rather than expected utility. This ranking represents a tradeoff between the total expected utility, which may accumulate from multiple goals, and the likelihood of achieving a particular subgoal.

Algorithm: *ResolveThreats*. This algorithm recursively resolves causal link threats, as in traditional partial order planning. Specifically, a causal link is of the form $T1$-$(C1 = ValueX)\rightarrow T2$, meaning task instance $T1$ achieves condition $C1 = ValueX$ as a precondition for task instance $T2$. A causal link threat occurs when another task instance, $T3$, has an effect of $C1 = ValueY$, where $ValueX \neq ValueY$, and is not ordered (by the current set of causal, data, and threat links) with respect to $T1$ and $T2$. To resolve this threat, $T3$ must be ordered either before $T1$ (demotion) or after $T2$ (promotion). *ResolveThreats* thus attempts to recursively resolve all causal link threats by promotion or demotion.

Algorithm: *Schedule*. With this algorithm, SA-POP moves from partial order planning to scheduling that meets stated resource requirements. SA-POP first determines the change in potential resource usage for each implementation (from the task map) of a task instance, given current rankings from the precedence graph. The *resource impact score* of an implementation is the sum across all resources of the percentage of resource capacity that would be utilized if all potentially overlapping task instances were to be executed concurrently. The implementation with the least impact on potential resource availability, as measured by the resource impact score, is chosen to implement the task instance, which is analogous to the least constraining value heuristic often used in general constraint satisfaction.

SA-POP also uses Laborie's energy precedence and balance constraint propagation techniques [7] modified for planning in DRE systems. These techniques are largely complementary and apply to different precedence sets with respect to a given task instance. The energy precedence constraint propagation can constrict time windows (and consequently derive more accurate rankings), even with relatively loose time windows that are prevalent early in planning. It applies to a task instance's start (end) time window based on the resource usage of all other task instances in its *Before* (*After*) precedence set.

SA-POP's balance constraint propagation applies to a task instance based on the other task instances in its *Unranked* and *Simultaneous* precedence sets. With the discrete resources, precedence graph, and links used by SA-POP, the constraint propagation differs from the Laborie calculations [7] with one major simplification: for discrete resources, only the start and end events of (potentially) overlapping task instances (*Unranked* and *Simultaneous* precedence sets) must be considered in the balance constraint. With this simplification, SA-POP uses Laborie's balance constraint propagation [7] to constrict time windows, impose necessary scheduling links, and detect irresolvable resource violations.

Algorithm: *ResolveRes*. This algorithm implements SA-POP's search for resolutions to potential resource violations. SA-POP employs two significant simplifications in the calculation of resource levels for events: (1) a potential resource conflict can only be resolved by imposing an ordering constraint (scheduling link) between two task instances (*i.e.*, by ordering an end event before a start event) and (2) given (1), only a worst case (minimum) resource level and best case (maximum) resource level need be calculated for each task instance (corresponding to the level when its start event occurs).

The heuristic for choosing the most significant resource violations is provided by a task instance criticality measure: $crit(x) = max(0, -L_{min}(x))/(L_{max}(x) - L_{min}(x)Q\Delta t_{start}(x)$ where, L is a resource level, Q is a resource capacity, and Δt is the length of a time window. After choosing the most critical task instance, x, a set of task instances from $Unranked(x)$ that can be ordered before x is chosen to reduce the criticality of x below the specified threshold. The heuristic for choosing these task instances is provided by preferring those with highest pair-wise criticality values given by: $crit(x, y) = -commit(y, x)/R(y)$ where, R is a resource usage value, and $commit(y, x)$ is a measure of the commitment implied by ordering the end event of y before the start event of x as defined in [7]. This heuristic provides a least commitment strategy (consistent with SA-POP's preference for minimally constrained operational strings) by balancing the preference for low commitment with the preference for high reduction in potential resource violations. With these algorithms, SA-POP employs backtracking whenever an irresolvable resource violation is discovered, or an attempt is made to impose a link inconsistent with the rankings in the precedence graph.

Table 2 is an example description of task map with implementations for the tasks in 2. In this scenario, C12 is

Task	Implementation	Resource Usage
A1	Impl1	1
A2	Impl2	1
A3	Impl3	1
A4	Impl4	1
A5	Impl5	1
A6	Impl6	2
A7	Impl7	4
A8	Impl8	1
A9	Impl9	5
A10	Impl10	1
A11	Impl11	1

Table 2. Example task map

a data condition, implying that A8 produces a data stream that can be consumed by A10 and/or A11. Using the expected utilities calculated for experiment 1 in the previous section, and a single resource of capacity 5 units for the system, Figure 5 illustrates the operational string generated by SA-POP. In this figure, the dashed arrow between A7

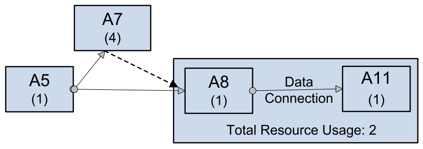

Figure 5. Operational string (capacity 5)

and A8 indicates a scheduling link, while other arrows indicate the causal links and the data link between A8 and A11. Because A8 and A11 are data manipulation tasks and no additional time constraints were imposed, they operate continuously until the end of the operational string's execution. Initially A7 would be unranked with respect to A8 and A11. Their combined resource usage of 2 and A7's resource usage of 4, however, would violate system resource capacities if they operated concurrently. Therefore, SA-POP imposes the scheduling link between A7 and A8 to ensure resource constraints are honored.

To illustrate the potential trade-off between expected utility and resource constraints, consider a similar system with the same goals but with a resource capacity of only 3 units. Figure 6 shows the operational string generated by SA-POP in this scenario. The tighter resource constraints

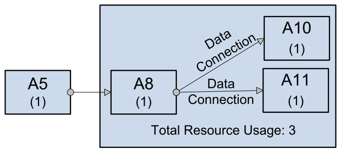

Figure 6. Operational string (capacity 3)

do not allow the inclusion of A7 to achieve C14, so SA-POP is forced to use the lower expected utility task A10 due to the limited resource availability.

4. Resource Allocation and Control Engine

The architecture of RACE and its interplay with SA-POP is illustrated in Figure 1. RACE performs autonomous resource (re)allocation and (re)configuration of QoS settings of components that are part of the operational strings generated by SA-POP such that the QoS requirements of the operational strings are met. RACE is built atop of CIAO and DAnCE, which are open-source (see `www.dre.vanderbilt.edu`) implementations of the OMG Lightweight CCM [12], Deployment and Configuration (D&C) [11], and Real-time CORBA [10] specifications. RACE provides a range of resource allocation and control algorithms that use middleware deployment and configuration mechanisms to allocate resources to operational strings and control system performance after operational strings have been deployed. In particular, it uses *Resource Monitors* and *ApplicationQoSMonitors*, which are implemented as CCM components, to track system resource utilization and application QoS respectively.

RACE's algorithms determine how to (re)deploy an application specified by operational strings and ensure desired QoS requirements are met, while maintaining resource utilization within desired bounds at all times. The allocation algorithms determine the initial component deployment by determining the best mapping of these components to the appropriate target nodes based on the availability of system resources. For example, an allocation algorithm could apportion CPU resources to components in such a way that avoids saturating these resources. Likewise, RACE's control algorithms adapt the execution of an operational strings' components at runtime in response to changing environments and variations in resource availability and/or demand. For example, a control algorithm could (1) modify an application's current operating mode, (2) dynamically update component implementations, and/or (3) redeploy all or part of an operational string's components to other target nodes to meet end-to-end QoS requirements.

RACE uses mechanisms provided by the underlying middleware to perform the allocation and control decisions made by its algorithms. For example, RACE uses standard mechanisms defined by the Lightweight CORBA Component Model (CCM) [12] to (1) (re)deploy and (re)configure application components, (2) transition application components from idle states to operational states and monitor the performance of the DRE system, and (3) modify components and/or operational strings to realize the adaptation decisions of control algorithms.

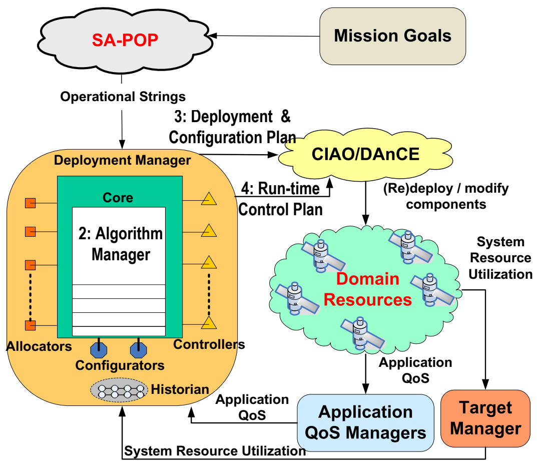

Figure 7. Architecture of RACE

As shown in Figure 7 the RACE architecture consists of the following entities that are implemented as CCM components using CIAO and deployed via DAnCE:

Resource Monitors are CCM components that track resource utilization in a domain. One or more *Resource-Monitors* are associated with each domain resource, such as CPU and memory utilization monitors on each node and network bandwidth utilization monitors on interconnects.

ApplicationQoSMonitors are CCM components that track the performance of application components by observing QoS properties, such as throughput and latency. One or more *ApplicationQoSMonitors* are associated with each type of application component.

The **TargetManager** [14] is a CCM component defined in the D&C specification [11] that receives periodic resource utilization updates from *ResourceMonitors* within a domain. It uses these updates to track resource usage of all resources within the domain. The *TargetManager* provides a standard interface for retrieving information pertaining to resource consumption of each component or assembly in the domain, as well as the domain's overall resource utilization.

The **DeploymentManager** is an assembly of CCM components that encapsulates and coordinates one or more allocation and control algorithms. This manager deploys assemblies by allocating resources to individual components in an assembly. After assemblies are deployed, the *DeploymentManager* manages the performance of (1) operational strings and (2) domain resource utilization. This manager ensures desired performance of the operational strings by performing the following actions to the components that make up the operational strings: (1) (re)allocating resources to the component, (2) modifying component parameters such as execution mode, and/or (3) dynamic replacing the component implementations.

5. Discussion and lessons learned

This section summarizes our experiences combining the decision-theoretic, resource-constrained planning of SA-POP with the component allocation and runtime management of RACE to produce an efficient and scalable architecture for autonomous operation of DRE systems in dynamic and uncertain domains. SA-POP produces partial-order plans that contain sufficient information to be instantiated with parameterized component implementations that do not violate coarse-grained resource constraints.

In the MMS system, for example, an instantiation of SA-POP on each satellite considers the computational resources, such as CPU, memory, and communication bandwidth to be monolithic, discrete resources. In actuality, there are multiple nodes with individual CPU and memory capacities within each satellite. In general, each task only uses a small fraction of these resources, so the course-grained resource constraints used by SA-POP helps ensure that RACE can find valid deployments for components on the real node resources.

Through the association of multiple functionally equivalent implementations for each task in the task map, RACE can find valid (re)allocations by substituting the original task components suggested by SA-POP with ones that are more resource friendly under the current conditions. In the unusual case that no such allocation is possible, RACE provides feedback to SA-POP indicating its failure to find a valid allocation due to one or more resource constraints. If this occurs, SA-POP generates a new operational string that uses less resources, but has lower expected utility, *without* requiring a repetition of the spreading activation.

Autonomous operation of satellites with limited computing capacity requires efficient algorithms to handle the combinatorial problems of planning, scheduling, and allocation. The loose coupling of SA-POP and RACE through a feedback loop, enables operational string generation as a search through a smaller space of potential resource-committed plans. The search is computationally less intensive than if resources were considered at the fine-grained node level.

Similarly, RACE does not have to consider the cascading task choices of planning to find a valid allocation, so its search space is also limited to a manageable size. Moreover, SA-POP only considers the *feasibility* of resource allocation in generating operational strings, while RACE can consider the harder resource *optimization* problem, but limits it to a given operational string. The limited size and complexity of the search spaces used in SA-POP and RACE, as well as the flexibility afforded by the task map, yields an architecture that can operate with limited computational resources, while scaling to relatively large planning and allocation problems without becoming intractable.

In generating the operational string from mission goals, SA-POP takes into account domain uncertainty by preferring operational strings of high expected utility. Rather than attempting the often intractable problem of finding operational strings with the highest overall expected utility, SA-POP's generates operational strings using a greedy approximation algorithm. The greedy choice of high expected utility tasks still yields a robust application as specified by the resulting operational string, but does not require the much greater search time needed to find the optimal solution.

For individual satellites to operate autonomously, they must be able to recognize and react to changes in local conditions. To this end, RACE monitors application performance and domain resource utilization using its *Application Monitors* and *Resource Monitors* after operational string deployment. If the performance of an operational string falls below its QoS requirement, RACE's control algorithms take corrective actions to achieve the specified QoS requirement.

For example, a control algorithm could (1) modify input parameters of one or more parameterized components of the operational string, (2) dynamically update task implementations from the choices available in the task map, and/or (3) redeploy all or part of an application's components to other target nodes to meet end-to-end QoS requirements. These actions help ensure that the QoS requirements of each operational string are met and resource utilization is maintained within specified bounds. If these control adaptations can not correct/prevent a QoS or resource violation, however, RACE notifies SA-POP, triggering replanning.

In addition to varying levels of resource utilization, runtime changes can occur in the environmental/system conditions represented in the task network. RACE continuously monitors these conditions and provides feedback on changes to SA-POP. SA-POP uses this information to incrementally update the probability values of conditions in the network, running forward propagation as necessary. Most changes correspond to the expected behavior of applications specified by operational strings. When a critical, unexpected change does occur, it can be handled more quickly because task network is updated. Critical changes are those that render the current application deployment nonfunctional for the achievement of some mission goal(s). As in the case of resource shortages, SA-POP performs plan repair by continuing operational string extraction with an open condition corresponding to the changed condition.

Revisions to mission goals, *e.g.*, due to onboard data analysis or revisions from mission scientists on the ground, are other runtime changes that may require modifications to deployed applications. The new/changed utility values for goals are inserted into the task network and the spreading activation mechanism is used to update it. These changes generally occur only for a small subset of the mission goals and thus only need be propagated through a relatively small portion of the full network. Moreover, only backpropagation of utility is necessary since probability values already forward propagated through the network are unchanged.

With the updated task network, a new operational string is generated using the same process described in Section 3.2. In this case, the operational string generation usually takes much longer than for plan repair because it must be completely regenerated in order to take advantage of the changed expected utilities. Fortunately, revised mission goals rarely render the current application deployment nonfunctional for all goals. In fact, unless the goals have changed drastically, the current operational string is probably still of high utility. As such, an immediate response to goal changes is not as critical as in the cases necessitating plan repair, so the time to extract a completely new operational string is insignificant in practice.

6. Concluding remarks

The paper described how we combined our SA-POP decision-theoretic planner for dynamic (re)planning with resource constraints under uncertainty with our RACE framework for resource allocation and control in autonomous and/or semi-autonomous DRE systems. We detailed SA-POP's spreading activation structure, which is a mechanism for determining the potential value of tasks using a decision-theoretic scheme, and our algorithm for generating operational strings based on expected utilities and resource constraints.

Empirical evaluation of our algorithm in the context of RACE demonstrated the effectiveness of our approach, even with the relatively limited resources available to individual elements of a DRE system. Our experiments showed how SA-POP and RACE can together facilitate autonomous operation by responding to dynamic changes through (re)planning of task sequences and the (re)deployment/(re)configuration of components. RACE and SA-POP are open-source software that can be obtained from `deuce.doc.wustl.edu/Download.html` as part of the CIAO middleware.

References

[1] S. Bagchi, G. Biswas, and K. Kawamura. Task Planning under Uncertainty using a Spreading Activation Network. *IEEE Transactions on Systems, Man, and Cybernetics*, 30(6):639–650, Nov. 2000.

[2] P. Baptiste and C. L. Pape. Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling. In *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group*, 1996.

[3] S. Curtis. The Magnetospheric Multiscale Mission...Resolving Fundamental Processes in Space Plasmas. *NASA STI/Recon Technical Report N*, pages 48257–+, Dec. 1999.

[4] X. Gu and K. Nahrstedt. Dynamic QoS-Aware Multimedia Service Configuration in Ubiquitous Computing Environments. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, 2002.

[5] G. T. Heineman and B. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.

[6] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[7] P. Laborie. Algorithms for Propagating Resource Constraints in AI Planning and Scheduling: Existing Approaches and New Results. *Artif. Intell.*, 143(2):151–188, 2003.

[8] M. Mikic-Rakic, S. Malek, and N. Medvidovic. Improving Availability in Large, Distributed Component-Based Systems Via Redeployment. In *3rd International Working Conference on Component Deployment (CD 2005)*, Grenoble, France, 2005.

[9] X. Nguyen and S. Kambhampati. Reviving Partial Order Planning. In B. Nebel, editor, *IJCAI*, pages 459–466. Morgan Kaufmann, 2001.

[10] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/05-01-04 edition, Aug. 2002.

[11] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.

[12] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[13] C. L. Pape. Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.

[14] N. Roy, N. Shankaran, and D. C. Schmidt. Bulls-Eye: A Resource Provisioning Service for Enterprise Distributed Real-time and Embedded Systems. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications*, Montpellier, France, Oct/Nov 2006.

[15] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. *CrossTalk - The Journal of Defense Software Engineering*, Nov. 2001.

[16] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, and G. Duzan. Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004.

[17] D. C. Sharp and W. C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[18] D. Smith, J. Frank, and A. Jonsson. Bridging the Gap Between Planning and Scheduling. *Knowledge Engineering Review*, 15(1):61–94, 2000.

[19] B. Srivastava and S. Kambhampati. Scaling up Planning by Teasing out Resource Scheduling. In S. Biundo and M. Fox, editors, *ECP*, volume 1809 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 1999.