Planning in Interplanetary Space: Theory and Practice*

Ari K. Jónsson and **Paul H. Morris** and **Nicola Muscettola** and **Kanna Rajan**
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035-1000,
{jonsson,pmorris,mus,kanna}@ptolemy.arc.nasa.gov

Ben Smith
Jet Propulsion Laboratory
Pasadena, CA 91109-8099
smith@aig.jpl.nasa.gov

Abstract

On May 17th 1999, NASA activated for the first time an AI-based planner/scheduler running on the flight processor of a spacecraft. This was part of the Remote Agent Experiment (RAX), a demonstration of closed-loop planning and execution, and model-based state inference and failure recovery. This paper describes the RAX Planner/Scheduler (RAX-PS), both in terms of the underlying planning framework and in terms of the fielded planner. RAX-PS plans are networks of constraints, built incrementally by consulting a model of the dynamics of the spacecraft. The RAX-PS planning procedure is formally well defined and can be proved to be complete. RAX-PS generates plans that are temporally flexible, allowing the execution system to adjust to actual plan execution conditions without breaking the plan. The practical aspect, developing a mission critical application, required paying attention to important engineering issues such as the design of methods for programmable search control, knowledge acquisition and planner validation. The result was a system capable of building concurrent plans with over a hundred tasks within the performance requirements of operational, mission-critical software.

Introduction

During the week of May 17th 1999, the Remote Agent became the first autonomous closed-loop software to control a spacecraft during a mission. This was done as part of a unique technology validation experiment, during which the Remote Agent took control of NASA's New Millennium Deep Space One spacecraft (Muscettola *et al.* 1998; Bernard *et al.* 1999a; 1999b). The experiment successfully demonstrated the applicability of closed-loop planning and execution, and the use of model-based state inference and failure recovery.

As one of the components of the autonomous control system, the on-board Remote Agent Experiment

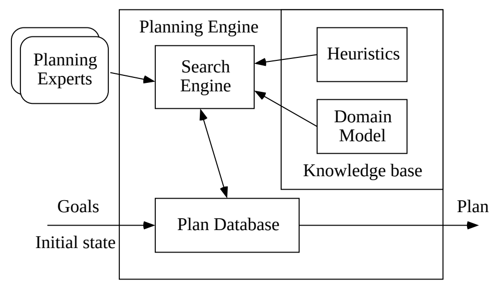

Figure 1: The Planner/Scheduler architecture

Planner/Scheduler (RAX-PS) drove the high-level goal-oriented commanding of the spacecraft. This involved generating plans that could safely be executed on board the spacecraft to achieve the specified high-level goals. Such plans had to account for on-board activities having different durations, requiring resources, and giving rise to subgoal activities, all while satisfying complex flight safety rules about activity interactions.

In this paper, we describe the Remote Agent Experiment Planner/Scheduler from both the theoretical and the practical perspectives. The architecture of the planning system is as shown in Figure 1. The *domain model* describes the dynamics of the system to which the planner is being applied – in this case, the Deep Space One spacecraft. A *plan request*, consisting of an initial state and a set of goals, initializes the *plan database*. The *search engine* then modifies the plan database to generate a complete valid *plan*, which is then sent to the execution agent. The *heuristics* and *planning experts* are not part of the core framework, but they are an integral part of the planning system that flew on board Deep Space One. The heuristics provide guidance to the search engine while the planning experts provide a uniform interface to external systems, such as attitude control systems, whose inputs the planner has to take into account.

*Authors in alphabetical order.

Theory

The RAX-PS system is based on a well-defined framework for planning and scheduling that, in many ways, differs significantly from classical STRIPS planning. For instance:

- Actions can occur concurrently and can have different durations.
- Goals can include time and maintenance conditions.

In this section, we will describe the PS framework from a theoretical perspective. We start out by describing how parallel activities are defined in the framework, how domain rules are specified, and what candidate plans are. We then go on to describe the semantics of candidate plans, from the point of view of plan execution, and derive a realistic definition of what is a valid plan. Finally, we present the planning process for this framework and prove that it is complete.

Tokens, Timelines and State Variables

To reason about concurrency and temporal extent, action instances and states are described in terms of temporal intervals that are linked by constraints. This approach has been called constraint-based interval planning (Smith, Frank, & Jónsson 2000), and has been used by various planners, including INOVA (Tate 1996) and IxTeT (Ghallab & Laruelle 1994). However, although our approach builds on constraint-based interval planning, there are significant differences. Among those are:

- The use of timelines to model and reason about concurrent activities
- The elimination of any distinction between actions and fluents
- The greater expressiveness of domain constraints

Humans find it natural to view the world in terms of interacting objects and their attributes. In planning, we are concerned with attributes whose states change over time. Such attributes are called *state variables*. The history of states for a state variable over a period of time is called a *timeline*. Figure 2 shows `Engine` and `Attitude` state variables, and portions of the associated timelines for a spacecraft application (the attitude of a spacecraft is its orientation in space). Between periods of idleness, the engine is thrusting in a given direction B. During this period, to achieve the correct thrust vector, the spacecraft attitude must be maintained so that it points in direction B. The turn actions change the attitude of the spacecraft.

In classical planning (Fikes & Nilsson 1971; McAllester & Rosenblit 1991), and earlier interval planning, there is a dichotomy between fluents and actions. The former specify states, and the latter specify transitions between them. In terms of interval planning, this has resulted in intervals describing only actions, and fluent values being implicit. However, this distinction is not always clear, or even useful. For example, in a spacecraft domain, thrusting in a direction `P` can either be regarded as a state that implies pointing towards `P` or an action with pointing towards `P` as a precondition. Moreover, during execution, the persistence of fluent values over temporal intervals may be actively enforced by maintaining and verifying the value. For these and other reasons, we make no distinction between fluents and actions in this planning approach, and use the same construct to describe both fluents and actions.

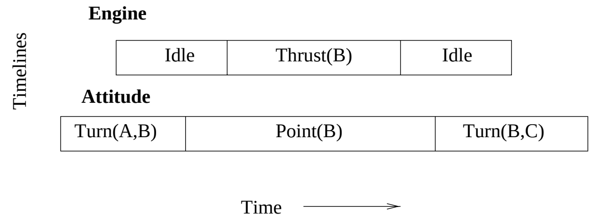

Figure 2: Plans as Parallel Timelines.

From the point of view of execution, a state variable represents a single thread in the execution of a concurrent system. At any given time, each thread can be executing a single *procedure* P. A procedure P has n_P parameters ($n_P \geq 0$), each with a specified type. Each state variable is also typed, i.e., there is a mapping $Procs : \mathcal{S} \rightarrow 2^{\Pi}$, where $\mathcal{S}$ is the set of state variables and Π is the set of all possible procedures. Given a state variable σ, $Procs(\sigma)$ specifies the procedures that can possibly be executed on σ.

Thus, a timeline consists of a sequence of intervals, each of which involves a single procedure. We may think of the interval and its procedure as being a structural unit, called a *token*, that has been placed on the timeline. Although each token resides on a definite timeline in the final plan, the appropriate timeline for a token may be undetermined for a while during planning. We refer to a token that is not yet on a timeline as a *floating* token.

A token describes a procedure invocation, the state variables on which it can occur, the parameter values of the procedure, and the time values defining the interval. To allow the specification of multiple values, e.g, to express a range of possible start times, variables are used to specify parameter, start and end time values for a token. As a result, a token T is a tuple $\langle v, P(\vec{x}_P), s, e\rangle$, where v is a variable denoting a state variable, P is the name of a procedure (satisfying $P \in Procs(v)$), the elements of $\vec{x}_P$ are variables that denote the parameters of the procedure (restricted to their types), and s and e are numeric variables indicating the start and end times respectively (satisfying $s \leq e$).

Each of the token variables, including the parameter variables, has a domain of values assigned to it. The variables may also participate in constraints that specify which value combinations are valid. For example, consider a token representing a camera taking a picture, where one parameter indicates the brightness level of the target object and another parameter specifies the choice of camera filter. Since the duration of the picture-taking token depends on the brightness level and the filter choice, a constraint links the start and end times with these parameters.

A more general notion of token was used in the RAX-PS for certain specialized purposes. This more general form, called a *constraint token*, is associated with more than a single procedure: it corresponds to a sequence of invocations, where each invocation is drawn from a specified set of procedures. The actual invocation sequence is determined during execution. Constraint tokens can be allowed to overlap as long as each overlap permits at least one valid procedure invocation. With this generalization, timelines can be used to represent resource usage. Each constraint token represents a resource demand. The combinations of overlapping demands must not exceed the available resource. Each intersected region of overlapping demand determines a procedure that assigns the resource, and checks that availability is not exceeded. This approach was used to model and keep track of power usage in the Remote Agent Experiment. Unfortunately, limited space prevents us from covering this generalization in detail in this article.

Domain Constraints

In a complex system, procedures cannot be invoked arbitrarily. A procedure call might work only after another procedure has completed, or it might need to be executed in parallel with a procedure on a different thread. For example, a procedure to turn from A to B can only occur after a procedure that has maintained the attitude at A, and it should precede a procedure that maintains the attitude at B. Similarly, a thrusting procedure can only be executed while another procedure maintains the correct spacecraft attitude.

To specify such constraints, each ground token, $T = \langle v, P(\vec{x}_P), s, e\rangle$, has a configuration constraint $G_T(v, \vec{x}_P, s, e)$, which we call a *compatibility*. It determines the necessary correlation with other procedure invocations in a legal plan, i.e., which procedures must precede, follow, be co-temporal, etc. Since a given procedure invocation may be supported by different configurations, a compatibility is a disjunction of constraints. Therefore, we define $G_T(v, \vec{x}_P, s, e)$ in terms of pairwise constraints between tokens, organized into a disjunctive normal form:

$$G_T(v, \vec{x}_P, s, e) = \Gamma_1^T \vee \cdots \vee \Gamma_n^T$$

Compatibilities also specify which procedure invocations are permitted; if the disjunction is empty, the procedure invocation is not valid in any configuration.

Each Γ_i^T is a conjunction of *subgoals* $\wedge_j \Gamma_{i,j}^T$ with the following form.

$$\Gamma_{i,j}^T = \exists T_j \gamma_{i,j}^T(v, \vec{x}_P, s, e, v_j, \vec{z}_{P_j}, s_j, e_j)$$

where T_j is a token $\langle v_j, P_j(\vec{z}_{P_j}), s_j, e_j\rangle$ and $\gamma_{i,j}^T$ is a constraint on the values of the variables of the two tokens involved.

In general $\gamma_{i,j}^T$ may take any form that appropriately specifies the relation between the two tokens. In practice, $\gamma_{i,j}^T$ is structured to limit its expressiveness and make planning and constraint propagation computationally efficient. In the RAX-PS framework, $\gamma_{i,j}^T$ is limited to conjunctions of:

- Equality (codesignation) constraints between parameter variables of different tokens.
- Simple temporal constraints on the start and end variables. These are specified in terms of metric versions of Allen's temporal algebra relations (Allen 1984); `before`, `after`, `meets`, `met-by`, etc. Each relation gives rise to a bound on the distance between two temporal variables. This bound can be expressed as a function of the start and end variables of T and T_j.

Subgoal constraints must guarantee that each state variable is always either executing a procedure or instantaneously switching between procedure invocations. This means that each Γ_i^T contains a *predecessor*, i.e., a requirement for a T_j on the same state variable as T, such that T `met_by` T_j. Similarly, each Γ_i^T must specify a *successor*.

The concept of subgoals generalizes the notion of preconditions and effects in classical planning. For example, ADD effects can be enforced by using `meets` subgoals while deleted preconditions correspond to `met_by` subgoals. Preconditions that are not affected by the action can be represented by `contained_by` subgoals.

In principle, a different compatibility may apply to each ground procedure invocation. In practice, a large number of invocations share the same constraints. For example, the process of executing an attitude turn is the same irrespective of where and when the turn starts or ends. Moreover, determining the set of applicable compatibilities must be done efficiently during the planning process. Since RAX-PS can reason about flexible tokens where variables have not been assigned single values, this is accomplished by indexing compatibilities hierarchically. The mechanism that is illustrated in Figure 3.

The basic idea is to associate compatibilities with sets that can be described as the Cartesian product of token variable domain subsets. This allows the planner to map from tokens to relevant compatibilities, by pairwise comparing domains. Procedure invocations that do not fall within one of the specified sets are not permitted. As an example, one set of constraints would be associated with minor attitude turns, while another set would be associated with large-scale attitude changes that require thrusters. Procedure invocations using the thrusters for small adjustments would therefore be excluded. In Figure 3, the round boxes (marked $V_i \rightarrow G_i$) represent compatibility associations. The compatibility G_i is applied to any token that falls within a set V_i. To see how this comes together, consider the straight boxes, marked $T1$ and $T2$, which represent tokens. It is easy to see and determine that $T1$ must be restricted to be within V_3 and that the compatibility G_3 is appli-

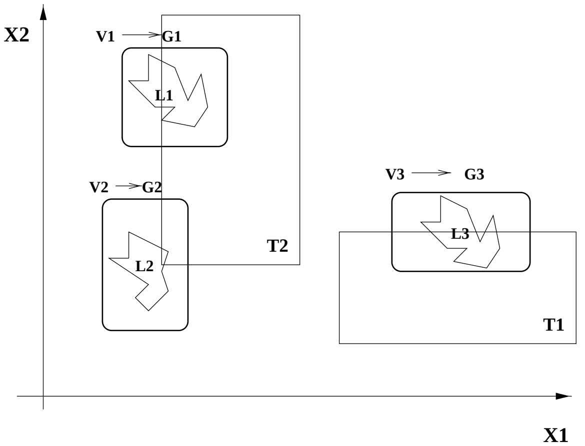

Figure 3: An illustration of the hierarchical indexing mechanism used in RAX-PS compatibilities.

cable to $T1$. The token $T2$ is too general to determine a single compatibility.

The use of Cartesian products, however, is too coarse-grained to specify valid procedure invocations. For example, the sets of possible origin and destination attitudes will relate to the sets of possible start and end times. Therefore, an additional constraint, the *local constraint* $L_{V_i}(v, \vec{x}_P, s, e)$, is associated to each set V_i[1]. The L_{V_i} are specified in terms of procedural constraints (Jónsson *et al.* 1999), which are an effective way to specify and enforce arbitrary constraints.

When a partially instantiated token intersects with only a single compatibility box, the instance of the corresponding local constraint L_P is automatically posted in the database and the compatibility associated with the box is made available for the planner to start satisfying appropriate sets of subgoals.

Plan Database

Having laid out the representation of the planning domain, we can now turn our attention to what the planner represents and reasons about. In RAX-PS, this is a data structure called the *plan database*. At the most basic level, the plan database represents 1) a current *candidate plan*, which is essentially a set of timelines containing interrelated tokens, and 2) a current set of decisions that need to be made.

In formal terms, a candidate plan consists of the following:

- a horizon (h_s, h_e), which is a pair of temporal values satisfying $-\infty \leq h_s < h_e \leq \infty$
- a timeline $\mathcal{T}_\sigma = (T_{\sigma_1}, \ldots, T_{\sigma_k})$, for each state variable, with tokens $T_i = \langle v, P_{\sigma_i}(\vec{x}), s, e \rangle$, such that each $P_{\sigma_i} \in Procs(\sigma)$
- ordering constraints $\{O_1, \ldots, O_K\}$, enforcing $h_s \leq e(T_{\sigma_1}) \leq s(T_{\sigma_2)} \leq \cdots \leq e(T_{\sigma_{k-1}}) \leq s(T_{\sigma_k}) \leq h_e$ for each timeline $\mathcal{T}_\sigma$
- a set of constraints $\{C_1, C_2, \ldots, C_N\}$, each relating sets of variables from one or more tokens; this includes temporal, equality and local procedural constraints

[1] This constraint is also used to limit the token variable domains to the set V_i

The constraints in a candidate plan give rise to a constraint network, consisting of the variables in the tokens and the constraints that link token variables in different ways. This network determines the set of all legal instantiations of the given tokens. As a result, any candidate plan that has an inconsistent underlying constraint network cannot be part of a valid plan.

An important aspect of the underlying constraint network is that it may be used to infer restrictions on possible values for token variables. This is done by *constraint propagation* (Mackworth & Freuder 1985) which is a method for eliminating values that can be proven not to appear in any solution to the constraint network. As a side-effect of removing values, the constraint propagation may also prove that no solution exists, by eliminating all values for some variable. Doing so implies that the candidate plan is invalid[2].

In addition to a candidate plan, the plan database may also contain a set of decisions that need to be made. A decision corresponds to a *flaw* in a candidate plan, an aspect of the candidate that may prevent it from being a complete and valid plan. In this framework, there are four types of flaws: *uninstantiated variables*, *floating tokens*, *open disjunctions of compatibilities*, and *unsatisfied compatibility subgoals*. Each flaw in the plan database gives rise to choices for how that flaw can be resolved. Resolving a flaw is a reasoning step that maps the given database to another database. Categorized by the types of flaws, the following is a list of the possible choices for resolving a flaw and the effect that this has on the plan database:

1. Variable restriction flaws are resolved by selecting a non-empty subset of the variable domain and restrict the variable to that domain. Effects:
   - If the restriction results in a token matching a unique compatibility specification, a new open disjunction flaw is added.
   - If the chosen domain is a singleton, the flaw is removed.
2. Floating token flaws are resolved by selecting two adjacent tokens on a timeline and inserting the floating token between them. Effects:
   - The floating token flaw is removed.
   - The ordering constraints implied by the insertion are added.
3. Open disjunction flaws are resolved by selecting one

[2] The converse is not necessarily true, as failing to find an empty domain does not guarantee the existence of a solution.

item in the disjunction and require that it be made true. Effects:

- The open disjunction flaw is removed.
- A set of unsatisfied subgoal flaws is added.
- Any implied constraints are added to the constraint set.

4. Unsatisfied subgoal flaws are resolved by either finding an existing token and using that to satisfy the subgoal, or by adding a new token to satisfy the subgoal. Effects:

- The unsatisfied subgoal flaw is removed.
- Resulting constraints are added.
- Implied ordering constraints are added, if a new token is generated.

It is important to note that it is not necessary to resolve all flaws in order to have a plan. For example, a valid plan might permit certain flexibility in token start and end times, which in turn means that some variable domains are not singletons. In most cases, however, we require that each token satisfy the applicable compatibility specification, i.e, that the subgoals from at least one of the disjunctions are satisfied. In that case, we say that the token is *fully supported*.

Plans and system behaviors

Based on the notions we have introduced here, we can now turn our attention to the semantics of a candidate plan, and the task of developing a formal definition of what a valid plan is. Traditionally, valid plans have been defined in abstract terms, based only on the candidate plan and the domain model. However, this approach is not realistic, as the validity of a plan in the real world is inherently tied to the mechanism that executes it. To address this, we start by discussing the basics of plan execution and then go on to derive a realistic definition of what constitutes a valid plan.

From the point of the executing agent (called the executive or EXEC) a plan is a concurrent program that is to be interpreted and executed in a dynamic system. Recall that the plan contains variables that specify how and under which circumstances procedures are to be instantiated. For variables that correspond to system values, such as the current time, the EXEC will sense actual system values, compare them with the values specified in the plan, and then determine which procedure should be executed next. If the EXEC fails to match sensed values with the values in the plan, the EXEC triggers a fault-protection response (e.g., put the system in a safe state and start taking recovery actions). The question of whether the EXEC succeeds in matching values and selecting a procedure invocation depends in part on how much reasoning the EXEC can perform for this purpose. That, in turn, depends both on how much reasoning the EXEC is capable of and how much time it has before the next invocation must be activated.

If an EXEC has no reasoning capabilities or is not permitted any time for deliberation, the execution process must be as simple as possible. For that purpose, the plan must be the simplest possible to interpret, i.e, all procedure calls should be fully specified and all invocation variables should be completely determined. Of course, this is the most brittle plan, since it provides only one possible match for each sensed values.

The simplest plans correspond exactly to single possible evolutions of the system. We will refer to them as *potential behaviors* of the system. Formally:

Definition 1 *A candidate plan is a* potential behavior *of the system if: (1) each token on each timeline is fully supported*[3]*; (2) all timelines fully cover the scheduling horizon* $[h_s, h_e]$*; and (3) all timeline token variables are bound to a single value.*

Consider now a candidate plan. In general there may be any number of gaps between timeline tokens, tokens may not be fully supported, and variables may be uninstantiated. In order to instantiate a single behavior, each flaw must be resolved successfully. For an execution agent with sufficient time and reasoning capabilities, such an under-specified plan might be a viable plan. In fact, the lack of commitment would allow the execution agent to choose the flaw resolutions that best fit the actual conditions during execution. The Remote Agent system took advantage of this by letting the EXEC map high-level tasks into low-level procedures, during execution. This freed the planner from generating low-level procedure calls, and allowed the executive to choose the low-level procedures that best fit the actual execution.

In general, executability depends on the execution agent in question. It depends primarily on two aspects; how flexible the candidate plan must be to cover possible system variations, and how restricted the candidate plan must be for the executive to identify whether it is executable. The latter is an important issue to consider, as making this determination can be as expensive as solving a planning problem.

To represent the abilities of a particular executive agent, we use a *plan identification function* f_I that identifies executable candidate plans, by mapping each possible candidate plan to one of the values of $\{T, F, ?\}$. The intent is that if a candidate $\mathcal{P}$ can be recognized as being executable, then $f_I(\mathcal{P}) = T$; if a candidate is recognized as not being executable, then $f_I(\mathcal{P}) = F$; and if executability cannot be determined, then $f_I(\mathcal{P}) =?$.

We permit a great deal of variation in how different executives respond to different candidate plans, but we do require that a plan identification function behaves consistently with respect to the two aspects mentioned above. For example, the function should not reject one

[3] In practice, we will always plan over a finite horizon. Therefore, this requirement is modified slightly for the start and end token of each timeline. In particular, the predecessor subgoals of T_{σ_1} are ignored, while the same applies to the successor subgoals of T_{σ_k}.

candidate on the basis of being too restrictive and then accept a restriction of that candidate. This leads us to the following formalization of what constitutes a plan identification function:

Definition 2 *A* plan identification function f_I *for a given execution agent is a function that maps the set of candidate plans to the extended truth value set* $\{T, F, ?\}$*, such that for any candidate plan* $\mathcal{P}$ *and any candidate plan* $\mathcal{Q}$ *that extends the candidate* $\mathcal{P}$*, we have:*

- *if* $f_I(\mathcal{P}) = F$ *then* $f_I(\mathcal{Q}) = F$
- *if* $f_I(\mathcal{P}) = T$*, then* $f_I(\mathcal{Q}) \in \{T, F\}$
- *if a token in* $\mathcal{P}$ *is not supported, then* $f_I(\mathcal{P}) = ?$

The last condition is not strictly necessary, as some executives are capable of solving planning problems, but in the interest of clarity, we will limit the execution agents to solving constraint satisfaction problems.

Using this notion of plan identification functions, we can now provide a realistic, formal definition of what constitutes a plan, namely:

Definition 3 *For a given executive, represented by a plan identification function* f_I*, a candidate plan* $\mathcal{P}$ *is a* plan *if and only if* $f_I(\mathcal{P}) = T$.

Planning process

We can now turn our attention to the plan generation process itself. The input to the planning process is an initial candidate plan, which includes an initialization token for each timeline, a set of floating tokens, and a set of constraints on the tokens in question. Together, these elements give rise to an initial plan database. The goal of the planning process is then to extend the given initial candidate to a complete valid plan. From the point of view of traditional planning, the initial plan database specifies both the initial state and the goals. In fact, our approach permits a much more expressive specification of goals. For example, we can request a spacecraft to take a specified sequences of pictures in parallel with providing a certain level of thrust.

The planning process we define is a framework that can be instantiated with different methods for controlling the search, selecting flaws, propagating constraints, etc. The planning process is a recursive function that non-deterministically selects a resolution for a flaw in the current plan database. An outline of the process is shown in Figure 4.

This planning process is clearly sound, as any resulting plan satisfies the given plan identification function. The planning process is also complete in the sense that if there is a plan, then a plan can be found. Furthermore, if a given initial candidate plan can be extended to some valid plan $\mathcal{P}$ (satisfying f_I), then the planning process can find some other valid plan (satisfying f_I) that can be extended to $\mathcal{P}$. A still stronger completeness criterion, that any given plan can be found, does not hold in general. The reason is that a lenient identification function f_I may return T even though the planning process has not addressed all remaining flaws.

```
plan (P,D)
  begin
    if f(P) = T
       return P
    else if f(P) = F
       return fail
    else
       given a flaw d from the flaw database D,
       choose a resolution res(d) for the flaw
       let (P',D') = apply res(d) to (P,D)
       return plan(P',D')
  end
```

Figure 4: The planning process. The plan database consists of the candidate plan P and the set of flaws D.

This highlights the importance of identifying properties of soundness and completeness for new planning frameworks such as this one.

Theorem 1 *Suppose a domain model, a plan identification function* f_I*, and an initial plan* $\mathcal{P}_0$ *are given. Let* $\mathcal{P}_T$ *be a valid plan (i.e.,* $f_I(\mathcal{P}_T) = T$*) that extends* $\mathcal{P}_0$*. Then, the planning process can generate a valid plan* $\mathcal{P}'$ *that extends* $\mathcal{P}_0$*, and can be extended to* $\mathcal{P}_T$.

Proof: The basic idea in the proof is to define an oracle that specifies how to resolve each flaw that the planning process may encounter, in order to get to a suitable plan $\mathcal{P}'$. This is straightforward to do, using the given target plan $\mathcal{P}_T$. The set of possible flaws is defined by the tokens that appear in the target plan; no other flaws need to be considered. For each flaw, the oracle specifies how it should be resolved:

- Variable domain restriction: Assign the variable the same domain it has in $\mathcal{P}_T$.
- Token insertion: Insert the free token so that it satisfies the ordering of tokens on that same timeline in $\mathcal{P}_T$.
- Compatibility choice: Choose any disjunction that is satisfied in $\mathcal{P}_T$.
- Subgoal satisfaction choice: Let T be a token that satisfies the subgoal in $\mathcal{P}_T$. If T already appears in the candidate plan, use that, otherwise, add T as a new token.

To show that this oracle will result in a suitable plan, we need to show that 1) all the flaws needed to arrive at the final plan eventually appear, 2) each step provides a candidate that can be extended to the final plan, and 3) the function f_I does not return F on any intermediate candidate plans. Steps 2) and 3) are straightforward. First, we show by induction that criterion 2) is maintained throughout the process. It is clearly true for $\mathcal{P}_0$. Each step of the planning process preserves it, since the chosen flaw resolution is always compatible with $\mathcal{P}_T$. Criterion 3) follows from the fact that if the plan identification function f_I returns F on some candidate plan $\mathcal{Q}$, it must return F on all candidate plans that extend $\mathcal{Q}$, which by 2) includes the final plan.

To prove step 1), that all the necessary flaws arise, we first note that it is sufficient to show that all tokens in $\mathcal{P}_T$ can be generated by the planning process. This will automatically give rise to the variable domain flaws, and the compatibility satisfaction flaws. To see that each token can be generated, recall that the initial candidate plan has an initialization token for each timeline. Also note that each compatibility specifies the possible successors (and predecessors) for a corresponding token. As a consequence, any given token on a timeline gives rise to a compatibility flaw that produces a succeeding token on that timeline. Straightforward induction then proves that this allows the planning process to generate all the tokens in $\mathcal{P}_T$. □

Practice

RAX PS extends the theoretical framework into a well-engineered system. The system had to operate under stringent performance and resource requirements. For example, the Deep Space 1 flight processor was a 25 MHz radiation-hardened RAD 6000 PowerPC processor with 32 MB memory available for the LISP image of the full Remote Agent. This performance is at least an order of magnitude worse than that of current desktop computing technology. Moreover, only 45% peak use of the CPU was available for RAX, the rest being used for the real-time flight software. The following sections describe the engineering aspects of the RAX PS system. First we describe the planning engine, the workhorse on which all development was founded. Then we describe the mechanism for search control used to fine-tune the planner. We also give information on the overall development process and on the methods of interaction with external software planning experts.

RAX PS planning engine

As follows from the previously discussed theory, producing a planner requires choosing a specific plan identification function f_I, a specific way to implement non-determinism and a flaw resolution strategy. In RAX PS we designed the planner in two steps. First we defined a basic planning engine, i.e., a general search procedure that would be theoretically complete. Then we designed a method to program the search engine and restrict the amount of search needed to find a solution. In this section we talk about the planning engine.

The first thing we need to clarify is what constitutes a desirable plan for the flight experiment. RAX plans are flexible only in the temporal dimension. More precisely, in a *temporally flexible plan* all variables must be bound to a single value, except the temporal variables (i.e., token start and end times, s and e). It is easy to see that under these assumptions the only un-instantiated constraint sub-network in the plan is a *simple temporal network* (Dechter, Meiri, & Pearl 1991). This means that the planner can use arc consistency to determine whether the plan contains any behavior and that the executive can adjust the flexible plan to actual execution conditions by using very fast incremental propagation (Tsamardinos, Muscettola, & Morris 1998). All of this is translated into a plan identification function f_I defined as follows: *When applied to a candidate plan, f_I checks its arc consistency. If the candidate is inconsistent, f_I returns F. If the candidate is arc consistent, f_I returns one of two values: T if the candidate is fully supported and all the non-temporal variables are grounded, and ? in any other case.*

To keep a balance between guaranteeing completeness and keeping the implementation as simple as possible, non-determinism was implemented as chronological backtracking. Also, the planner always returned the first plan found. Finally, the planning engine provided a default flaw selection strategy at any choice points of the backtrack search. This guaranteed that no underconstrained temporal variable flaw would ever be selected, while all other flaw selection and resolutions were made randomly.

Model size		
	State variables	18
	Procedure types	42
Plan size		
	Tokens	154
	Variables	288
	Constraints	232
Performance		
	Search nodes	649
	Search efficiency	64%

Table 1: Plan size and performance of RAX PS

Search control

By itself, the basic planning engine could not generate the plans needed for the flight experiment. However, RAX PS included additional search control mechanisms that allowed very localized backtracking. This is reflected in the the performance figures in Table 1, where search efficiency is measured as the ratio between the minimum number of search nodes needed and the total number explored.

Achieving this kind of performance was not easy and required a significant engineering effort. We outline the principal aspects of this effort in the rest of the section.

Flaw agenda management RAX PS made use of a programmable search controller rather than the default flaw selection strategy described before. Ideally, the "optimal" search controller is an oracle such as the one described before in the proof of completeness. Having advance knowledge of the plan $\mathcal{P}_T$, the oracle can select the correct solution choice without backtracking. In practice this is not possible and the control strategy can only make flaw resolution decisions on the basis of the partial plan developed so far. The search controller of RAX PS allows programming an approximate oracle as a list of search control rules. This list provides

```
(:subgoal
 (:master-match (Camera = Ready))
 (:slave-match  (Camera = Turning_On))
 (:method-priority ((:method :add)(:sort :asap))
                   ((:method :connect))
                   ((:method :defer)))
 (:priority 50))
```

Figure 5: Search control rules for unsatisfied subgoal

a prioritization of the flaws in a database and sorting strategies for the non-deterministic choices for each flaw selection. Figure 5 gives an example of a search control rule.

The rule applies to an unsatisfied subgoal flaw of a $\langle \texttt{Camera}, \texttt{Ready}, s, e\rangle$ token that requires a $\langle \texttt{Camera}, \texttt{Turning\_on}, s_k, e_k\rangle$ token. Note that in the DS1 model the `Camera` can reach a `Ready` state only immediately after the procedure `Turning_on` has been executed. Therefore, in this case, matching the token types in the subgoal is sufficient to uniquely identify it. When the priority value associated with the flaw is the minimum in the plan database, the planner will attempt to resolve the flaw by trying the resolution methods in order. In our case the planner will first try to `:add` a new token and try to insert it in the earliest possible timeline gap (using the standard sort method `:asap`). The last resolution method to try is to `:defer` the subgoal. When this happens, the plan database will automatically force start or end of the token to occur outside of the horizon h_s. In our case, the deferment method will only succeed if the `Ready` token is the first token on the timeline.

Search control engineering The rule language for the search controller is designed to be extremely flexible. It permits the introduction of new sorting methods, if the standard methods prove to be ineffective. Also, it is possible to prune both on solution methods (e.g., only `:connect` to satisfy a subgoal) and on resolution alternatives (e.g., try scheduling a token only as early as possible and fail if you cannot). Unfortunately, this meant that completeness could no longer be guaranteed. On the other hand it allowed for a finely tuned planner. Designing search control became therefore an exercise in trading off between scripting the planner's behavior and exploring the benefits of shallow backtracking when necessary. Here are some issues that needed to be addressed.

Interaction between model and heuristics: Ideally, it is desirable to keep domain model and search control methods completely separate. This is because constraints that describe the "physics" of the domain should only describe what is possible while search control should help in narrowing down what is desirable from what is possible. Moreover, declarative domain models are usually specified by domain experts (e.g., spacecraft systems engineers) not by problem solving experts (e.g., mission operators). Commingling structural domain information with problem solving methods can significantly complicate inspection and verification of the different modules of a planning system.

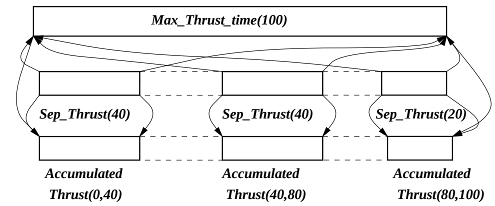

Figure 6: A plan fragment implementing thrust accumulation within a plan horizon

In our experience, however, such an ideal separation was difficult to achieve. Model specifications that were logically correct turned out to be very inefficient because they required the discovery of simple properties by extensive search (e.g., a token being the first of a sequence of tokens with the same procedure). The standard method used in RAX-PS was to define auxiliary token variables and use search control to enforce a specific value, which in turn would prune undesired alternatives through constraint propagation. Including the control information within the model caused a significant level of fragility in domain modeling, especially in the initial stages of the project when we still had a weak grasp on how to control the search.

Using global control information: RAX-PS search control requires rules to rely solely on local information. For example, a variable restriction rule can only rely on the information in the token to which the variable belongs. Sometimes, however, global information is needed to make control decisions. For example, in the DS1 the planner needs to schedule the correct amount of accumulated thrust within a planning horizon. This requires keeping track of the sum of the duration of each `Thrust` token in each candidate plan. New `Thrust` tokens will not be generated if the sum exceeds a given limit. The solution we adopted (Figure 6) was to appropriately program the domain model, so that the constraint propagation mechanisms could compute the global information. In particular, we included an extra `Thrust_Accumulation` timeline whose tokens effectively act like a global timer. Tokens on this timeline captured the possible start and end time ranges of each `Thrust` token by variable codesignations in subgoals. Local constraints then performed the summation and predecessor/successor subgoals propagated the sum to other `Thrust_Accumulation` subgoal tokens.

High-level control languages: The control rules described above can be thought of as an "assembly language" for search control; and the DS1 experience confirmed that programming in a low-level language is painful and error prone. However, this assembly language provides us with a strong foundation on which to build higher level control languages which are well founded and better capture the control knowledge of mission operators. The declarative semantics of the

Subsystem	*State Variables*	*Value Types*	*Constraints*	*Comments*
MICAS	Executable: 2 Health: 1	7	14	Models the health, mode and activity of the MICAS imaging camera. RAX demonstrates fault injection and recovery for this device as part of the 6 day scenario.
Navigation	Goal: 1 Executable: 1 Internal: 1	5	6	To schedule Orbit determination (OD) based on picture taking activity.
Propulsion & Thrust	Goal: 2 Executable: 1 Internal: 1	9	12	Based on thrust schedule generated by the NAV module, the planner generates plans to precisely activate the IPS in specific intervals based on constraints in the domain model and is the most complex set of timelines and subsystem controlled by the planner.
Attitude	Executable: 1 Health: 1	4	4	Enables the planner to schedule slews between constant pointing attitudes when the spacecraft maintains its panels towards the sun. The targets of the constant pointing attitudes are imaging targets, Earth (for communication) and thrust direction (for IPS thrusting).
Power Mgmt.	Goal: 1 Internal: 1	2	1	Allows the planner to ensure that adequate power is available when scheduling numerous activities simultaneously.
Executive	Goal: 1 Executable: 1	2	7	Allows modeling of low level sequences bypassing planner models giving Mission Ops the ability to run in sequencing mode with the RA.
Planner	Executable: 1	2	2	To schedule when the Executive can request the plan for the next horizon.
Mission	Goal: 1	2	2	Allows the Mission Manager and the planner to coordinate activities based on a series of scheduling horizons updatable by Mission Ops for the entire mission.

Figure 7: Timelines of the RAX domain model

domain model also opens up the possibility of automatically understanding dependencies that point to effective search control. The synthesized strategies can then be compiled into the low-level control rules. Work is currently in progress to explore the viability of such methods to alleviate the burden of control search programming.

Scenario-driven development and testing

We used a scenario-driven iterative refinement process to develop the domain model. Domain models were based on a fixed scenario. The scenario might involve a certain amount of thrust activity, communication windows and picture taking activity. When the scenario was changed, the fragility of the model was immediately apparent with the planner not converging within resource bounds. Testing, therefore, was a critical component in deployment.

The scenarios also drove planner testing. We developed several scenarios to cover the possible modifications to the baseline, and to exercise fault conditions. The scenarios were run automatically with a test harness. Automated verification tools reported cases where the planner failed to converge, or where the planner generated an incorrect plan. The testing process is described more fully in (Smith *et al.* 1999).

Interaction with Plan Experts

Quite often, legacy or specialized software external to the planner provides specialized knowledge about the development of plan fragments. We call such software *planning experts*. The RAX-PS framework provides a practical solution to the direct integration of this knowledge into the planning process. Experts can be wrapped into appropriate adaptors that present their products as if they were coming directly from the declarative model.

It is illustrative to describe in more detail the interaction between RAX-PS and the optical navigation (OPNAV) system. OPNAV was one of the revolutionary technologies validated by DS1. During the nominal mission, when RAX was not active, OPNAV periodically commanded taking pictures of beacon asteroids to triangulate the position of the spacecraft and estimate whether the spacecraft was on course. When RAX was active, OPNAV would simply provide a source of planning goals, i.e., the beacon asteroids to be imaged. RAX-PS would then plan the detailed activities needed.

The communication of goals from OPNAV to RAX-PS worked as follows. Before OPNAV could be invoked, the planner had to extend the plan enough to know at what time it wanted to perform imaging activities. At this point a search control rule would explicitly invoke OPNAV and as a result a set of floating tokens and relative temporal constraints would be deposited in the plan database. In principle the planner could reject any of them. However the constraints posted by OPNAV and the design of the search control rules ensured that under normal circumstances the planner would schedule the goals linearly in time (with higher priority goals scheduled first) and only start rejecting OPNAV goals when it ran out of time in the allotted temporal window.

Developing a principled interaction between planning systems and legacy control software is a topic of active research, and is very important for the practical acceptability of planning technology.

The Planner in Flight

The Remote Agent Experiment was conducted during the week of May 17th. The experiment achieved all of the technology validation objectives. However, it was not without surprises. The most notable occurred in the early morning of May 18th when the RAX team realized that Remote Agent had ceased to command the spacecraft while being otherwise healthy. In the next 10 hours the problem was diagnosed by analyzing telemetry data downlinked from the spacecraft and by inspecting the source code. The problem turned out to be a low probability deadlock condition due to a missing critical section in the EXEC code. Within the following 10 hours the RAX team developed a potential software patch, developed a completely new experimental scenario to complete the achievement of all Remote Agent validation objectives, and validated the scenario by running it on flight analog hardware. The new scenario was activated in the morning of May 21st. In spite of another problem in communication software external to the Remote Agent, the experiment completed around 14:00 PDT achieving 100% of the validation objectives.

RAX-PS performed flawlessly. Most importantly, without the planner the experiment could simply not have resumed after the interruption given the tight time constraints. Developing, testing and approving a new sequence of complex activities on a spacecraft usually requires several days (Rayman 1999). With a planner, a new mission scenario could be developed in less than an hour. Indeed, it is worth noting that most of the overnight testing and validation involved running the full 6 hours of the new scenario on the flight processor, in real time. In the end, a potentially catastrophic software fault turned out to be a unexpected showcase of how planning technology can robustify and reduce costs for future robotic space missions. Details of the actual flight run can be seen in (Bernard *et al.* 1999a) and (Nayak *et al.* 1999).

Conclusion

In this paper, we have presented an overview of the Remote Agent Experiment Planning/Scheduling system, both from theoretical and practical points of view. On the theoretical side, we described the underlying planning framework, which in many ways is different from traditional planning approaches. Among many other advantages, the framework provides the ability to plan concurrent activities that have different durations, and the expressiveness needed to plan for complex interacting goals, including maintenance goals. On the practical side, we discussed some of the problems that had to be solved during the implementation and testing of the planner as flight software, and presented our solutions to these problems.

Research and development of autonomous planning systems, capable of solving real problems, continues among the many scientists in the field. The work we have presented here is just another step in this development, but it is a step that has taken autonomous planning to interplanetary space.

Acknowledgments

The authors acknowledge the support of the complete Remote Agent team from NASA Ames and JPL. We would particularly like to thank Steve Chien, Scott Davies, Greg Rabideau and David Yan who contributed to the RAX-PS flight experience. We also thank Jeremy Frank, David E. Smith, and the anonymous reviewers for their comments.

References

Allen, J. 1984. Towards a general theory of action and time. *Artificial Intelligence* 23(2):123–154.

Bernard, D.; Dorais, G.; Gamble, E.; Kanefsky, B.; Kurien, J.; Man, G. K.; Millar, W.; Muscettola, N.; Nayak, P.; Rajan, K.; Rouquette, N.; Smith, B.; Taylor, W.; and Tung, Y.-W. 1999a. Spacecraft autonomy flight experience: The DS1 Remote Agent experiment. In *Proceedings of the AIAA Conference 1999, Albuquerque, New Mexico.*

Bernard, D. E.; Dorais, G. A.; Fry, C.; Jr., E. B. G.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P. P.; Pell, B.; Rajan, K.; Rouquette, N.; Smith, B.; and Williams, B. C. 1999b. Design of the Remote Agent experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference, 1999.*

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems.*

Jónsson, A. K.; Morris, P. H.; Muscettola, N.; and Rajan, K. 1999. Next generation Remote Agent planner. In *Proceedings of the Fifth International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS99).*

Mackworth, A. K., and Freuder, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence* 25:65–74.

McAllester, D., and Rosenblit, D. 1991. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 634–639.

Muscettola, N.; Nayak, P. P.; Pell, B.; and William, B. 1998. Remote Agent: To boldly go where no ai system has gone before. *Artificial Intelligence* 103(1-2):5–48.

Muscettola, N. 1994. HSTS: Integrated planning and scheduling. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling.* Morgan Kaufman. 169–212.

Nayak, P. P.; Bernard, D. E.; Dorais, G.; Jr., E. B. G.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Rajan, K.; Rouquette, N.; Smith, B. D.; Taylor, W.; and wen Tung, Y. 1999. Validating the DS1 Remote Agent experiment. In *Proceedings of the Fifth International Symposium on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS) 1999 ,Noordwijk, The Netherlands.*

Rayman, M. 1999. Personal communication with Marc Rayman, Chief Mission Manager DS1.

Smith, B.; Millar, W.; Dunphy, J.; wen Tung, Y.; Nayak, P. P.; Jr., E. B. G.; and Clark, M. 1999. Validation and verification of the Remote Agent for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference 1999, Snowmass, CO.*

Smith, D. E.; Frank, J.; and Jónsson, A. K. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1).

Tate, A. 1996. Representing plans as a set of constraints - the ⟨I-N-OVA⟩ model. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems.*

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast transformation of temporal plans for efficient execution. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, 254–261. Menlo Park: AAAI Press.