NEXT GENERATION REMOTE AGENT PLANNER

Ari K. Jónsson
RIACS
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035, USA
phone: +1 650 604 2799
fax: +1 650 604 3594
jonsson@ptolemy.arc.nasa.gov

Paul H. Morris
Caelum Research Corporation
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035, USA
phone: +1 650 604 4713
fax: +1 650 604 3594
pmorris@ptolemy.arc.nasa.gov

Nicola Muscettola
Recom Technologies
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035, USA
phone: +1 650 604 4744
fax: +1 650 604 3594
mus@ptolemy.arc.nasa.gov

Kanna Rajan
Caelum Research Corporation
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035, USA
phone: +1 650 604 0573
fax: +1 650 604 3594
kanna@ptolemy.arc.nasa.gov

ABSTRACT

In May 1999, as part of a unique technology validation experiment onboard the Deep Space One spacecraft, the Remote Agent became the first complete autonomous spacecraft control architecture to run as flight software onboard an active spacecraft. As one of the three components of the architecture, the Remote Agent Planner had the task of laying out the course of action to be taken, which included activities such as turning, thrusting, data gathering, and communicating.

Building on the successful approach developed for the Remote Agent Planner, the Next Generation Remote Agent Planner is a completely redesigned and reimplemented version of the planner. The new system provides all the key capabilities of the original planner, while adding functionality, improving performance and providing a modular and extendible implementation. The goal of this ongoing project is to develop a system that provides both a basis for future applications and a framework for further research in the area of autonomous planning for spacecraft.

In this article, we present an introductory overview of the Next Generation Remote Agent Planner. We present a new and simplified definition of the planning problem, describe the basics of the planning process, lay out the new system design and examine the functionality of the core reasoning module.

1. INTRODUCTION

The Remote Agent (Muscettola *et al.* 1998) is the first complete autonomous spacecraft control architecture to run as flight software onboard an active spacecraft. In a unique experiment in May of 1999, the Remote Agent was flight-validated onboard the Deep Space One spacecraft. During this experiment, the Remote Agent successfully generated complex plans which included thrusting of the Ion Propulsion System, slewing and taking pictures. The Remote Agent executed the generated plans safely, and correctly handled a number of injected faults during execution.

As discovered during the development of the Remote Agent Planner, the spacecraft domain provides a number of challenges that are typically not addressed in autonomous planning technology development:

- Activities are executed concurrently onboard the spacecraft, so a plan consists of concurrent activity sequences that can safely be executed in parallel.
- Resources, such as power, fuel, data storage, are strictly limited. A planner must guarantee that possibly concurrent activities in a plan will not exceed resource availability.
- Activities have complex interactions and constraints between them, and any plan generated by the planner must satisfy all constraints and take all interactions into account.

- Activity duration is often flexible. A planner must therefore be capable of reasoning about activities that only have bounds on their duration.

To meet these challenges, the Remote Agent Planner was based on an approach to planning that departs from the more classical planning approaches (Bylander 1994) in a number of ways. (1) The planner reasons about parallel activity sequences, each of which represents the changing state of some system attribute. (2) It can reason about activities that have flexible duration, while taking into account quantitative temporal constraints between them. (3) The goal of the planner is not to generate a fixed sequence, but rather to generate a plan description that is suitable for execution. (4) The planner handles a rich action representation language that can describe the complex activities of real-world systems. This language is also unique in that it eliminates the syntactic and semantic distinction between actions and steady-states. (5) The planner allows for a structured domain description language that is sufficiently expressive to describe the rules and interactions in complex real-world domains such as spacecraft.

The applicability of this approach to real-world planning problems was clearly demonstrated in the Remote Agent Experiment. Nonetheless, work continues on the development of the approach, both in terms of the underlying planning framework and in terms of the implemented planning system. The Next Generation Remote Agent Planner is the next step in this development, providing a simpler and clearer definition for the planning framework, and an enhanced, modular implementation of the planning system.

The simplified planning framework is derived directly from the framework underlying the original Remote Agent Planner. It is just as expressive as the original framework, but has been simplified by unifying concepts and simplifying the problem specifications. The implemented planner is also based on the original planner, but a number of interesting enhancements have been made. First of all, it is based on a new modular system design, aimed at making it easy to modify, maintain and enhance the different components that make up the system. Secondly, the interface that the core system provides to the top-level planner search engine has been significantly simplified. Whereas the original system was limited to backtracking search, the new framework, in conjunction with the simplified top-level interface, make it possible to utilize other, possibly more efficient, search techniques, such as repair-based search and dependency-directed search. Third, the new system includes a new constraint reasoning module that allows arbitrary procedural constraints to be used. This speeds up the constraint reasoning, which is a crucial part of the planning process, and eliminates previous limitations on the set of constraints that can be represented.

In this paper, we describe the simplified planning framework, and give an overview of the new implemented planning system. We first present the planning framework in an informal manner. We then describe the approach used to solve the planning problems, and give an overview of the new planning system. We continue by providing some details about the new constraint reasoning mechanism, and conclude by looking at what has been done and what is on the agenda.

2. THE PLANNING FRAMEWORK

In this section, we will describe the simplified planning framework, on which the Next Generation Remote Agent Planner is built. The planning framework defines the class of planning problems being solved, i.e, what the world looks like to the planner, and what constitutes a valid plan.

Let us start by looking at what the end result of the planning process should be, i.e, what constitutes a plan. Considering the planner as part of the Remote Agent system, a completed plan is a program or a recipe for what activities the Remote Agent Executive should perform and what states should be maintained. In classical flight software systems, such a plan consists of time-stamped tasks, each to be executed at the predetermined time. The problem with that approach is that it requires an explicit tradeoff to be made between robustness and efficiency. If the time allocated to a task is close to the estimated execution, any delay will result in failure. However, if the time allocated to the same task is much more than the estimated execution time, then time is wasted. To resolve this problem, the Remote Agent is capable of handling temporal flexibility in timepoints describing transitions such as going from the engine thrusting to the engine being off. This means that the start of one task can be tied to the completion of another task, minimizing the effect of any delays, while maintaining the robustness of the plan. The end result of this is that the generated plan is defined, not by fixed times for transition timepoints, but rather by bounds on those timepoints and temporal constraints between them. Figure 1 shows what a simplified, small plan might look like.

In order to define the planning framework, we must now specify what "activities" are, what temporal constraints are and what constitutes a valid plan. Since the exact set of activities and rules will depend on the environment in which the planner operates, the planner uses a description of the activities and rules in each environment. Such a description is called a *domain model*, as

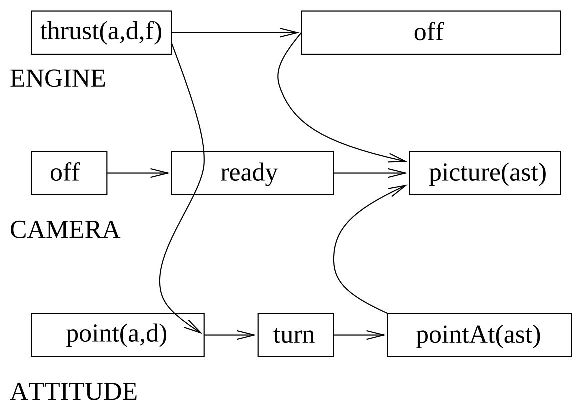

Figure 1: A simplified plan showing activities for engine, camera and attitude. Arrows show temporal constraints between transition timepoints.

it models the domain in which the planner is operating. Describing the planning framework is therefore largely a question of defining what a domain model is.

Many real-world systems, including spacecraft, can naturally be described in terms of components that at each point in time are in a certain state or performing a certain activity. For example, at any point in time, the attitude system can either be holding a specific attitude, or turning from one attitude to another. This natural approach to modeling real-world systems is mirrored in the planner, which plans by reasoning about how the states of such components can change over a given period of time. To generalize this, the basic concept in the domain model is an *attribute* which describes a part of the world that can change over time, e.g, the state of a spacecraft system component.

To specify an attribute, the set of possible values (representing states or activities) must be given. Since states and activities are often fairly complex, the attribute values are described in terms of *predicates* that can have multiple parameters. For example, the attribute value describing the state of holding a constant attitude must have the pointing coordinates as parameters, resulting in a predicate of the form `constantPointing(a,d)`, assuming equatorial coordinates.[1]

A predicate is defined by a unique *predicate name*, a sequence of *parameter domains* and optionally a set of *parameter constraints*, which limit the set of valid parameter value combinations. For an example of a predicate, let us consider an attribute describing the amount of data stored on the onboard data recording mechanism. A predicate describing data being recorded, aptly named `record`, has four parameters; the amount of data at the beginning of the activity, the rate at which the data is being collected, the duration of the activity, and the amount of data at the end. Each parameter takes a value from a given domain; for example, the start-data and the end-data parameters have values between 0 and M, where M is the maximum data storage capacity. Obviously, not all combinations of the possible parameter values give rise to a valid record activity description. Therefore, the final component of the predicate definition is the constraint that for any instantiation `record(s,r,d,e)`, the parameters must satisfy `s + rd = e`.

To structure the domain model, attributes are arranged together as components of model objects, which in turn are instances of model classes. This means that a *model class* is essentially a set of named attributes. For example, a class describing engine objects might have a fuel level attribute, an engine state attribute and a thrust attribute. The *model objects*, such as a specific engine, are then instances of these classes. This allows the same class definition to be used for multiple instances, e.g, in a spacecraft with multiple engines.

Having seen how the predicates describe the values that each attribute can take, let us now turn our attention to the interactions between different attributes. This interaction is the main complicating factor in real-world systems, as many configurations and sequences are either not possible or not safe. For an example of such interactions, let us consider a spacecraft that has an engine and a camera. Since the engine thrust causes vibrations, the camera cannot be taking pictures during the times the engine is thrusting. This leads to the constraint that whenever the camera is taking pictures, the engine must be off. Rephrasing this slightly, the constraint states that any continuous temporal interval where the camera is taking a picture must be contained within a continuous interval where the engine is off.

In order to be able to describe this containment and other relations between intervals, the planner uses quantitative temporal relations. There are twelve possible relations that come in pairs where one is the inverse of the other. The six temporal relations classes are:

- before, after
- startsBefore, startsAfter
- endsBefore, endsAfter
- startsBeforeEnd, endsAfterStart
- contains, containedBy
- parallels, paralleledBy

Quantitative bounds can be placed on the distance be-

[1] Technically, these are not predicates, as they do not evaluate to true or false by themselves. However, they can be viewed as shortcuts for the predicates representing that a given attribute has that particular compound value.

tween any two timepoints involved in the interval relation. For example, "before[10,20]" indicates that the first interval must end at least 10 and no more than 20 time units before the second one starts.

To specify rules, such as the one involving the engine and camera, we use a construct called a *configuration constraint*. In principle, a configuration constraint is defined for each possible instantiation of a predicate. Thus, each configuration constraint consists of a predicate instance (attribute value) v and a set of pairs $\{(\tau_1, V_1), \ldots, (\tau_k, V_k)\}$, where τ_i is a temporal relation and V_i is a set of instantiations of a predicate. The semantics of such a constraint are that for any interval I where an attribute has the value v, there must, for each $i \in \{1, \ldots, k\}$, be an interval J_i where an attribute has one of the values in V_i and the interval pair (I, J_i) satisfies the temporal constraint τ_i.

For an example of such a configuration constraint, let us write up the one for a camera taking a picture of a specific asteroid. In textual form, the configuration constraint can be specified as follows:

```
(camera == picture(asteroid))
    containedBy(engine == off)
    containedBy(attitude == pointAt(asteroid))
    before[0,0](camera == ready)
    after[0,0](camera == ready)
```

The "containedBy" relations specify that each of the engine-off and point-at-asteroid intervals must start no later than at the start of the picture-taking interval and end no earlier than when the picture-taking interval ends.[2] The "before[0,0]" and "after[0,0]" relations enforce that camera-ready intervals must immediately precede and follow the picture-taking interval. Figure 2 shows a graphical representation of this configuration constraint.

It should be noted that although configuration constraints are conceptually defined for each predicate instantiation, in practice, they are specified in the form of *configuration constraint schemata*. Such schemata specify patterns rather than instantiated attribute values, thus collapsing large sets of constraints into a single schema. The constraints are then instantiated from the schemata whenever sufficient information is available to determine that they are applicable to a given interval.

3. THE PLANNING PROCESS

The Next Generation Remote Agent planning process is based on representing and reasoning about the

[2]Not displaying the bounds is short-hand for the distance bounds being $[0, \infty]$.

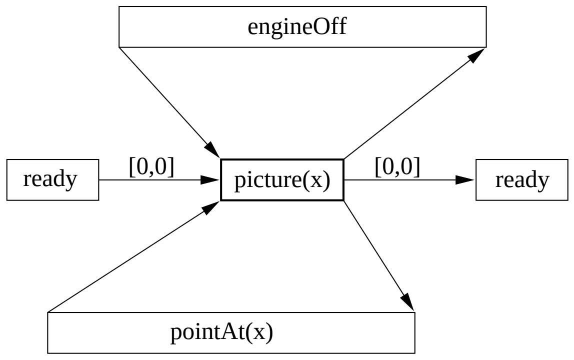

Figure 2: A graphical representation of a configuration constraint. The links indicate temporal constraints that limit the distance from one timepoint to another.

possible developments of each attribute over the time period for which the planner is planning. The goal of this reasoning process is to generate a plan consisting of a network of transitions between attribute values, such that all configuration constraints are satisfied.

The approach used by the planner is to generate and reason about structures called *tokens*. Each token represents a restriction on the set of values that an attribute may take over a specified temporal interval. A *value token* is a special type of token, having the additional restriction that the attribute must maintain a single value throughout the associated interval. Other types of tokens are used in the RA planner, such as constraint tokens which limit the attribute value to a given set, but do allow the attribute value to change during the interval. However, for clarity we will only consider value tokens in this paper. From here on, any reference to a token should therefore be read as referring to a value token.

The planner utilizes variables to represent the different elements of a token. This allows the planner to reason effectively about tokens and their interactions. As a result, a token consists of:

- A predicate name
- A variable representing the start time
- A variable representing the end time
- A variable representing the duration
- A set of parameter variables, one for each parameter to the predicate

In addition to the variables, any applicable parameter constraints are associated with a token, and so is a temporal constraint enforcing that the sum of the start time and the duration is equal to the end time.

Other temporal constraints may then link start and end timepoints from different tokens. These can stem from configuration constraints, or be instantiated as part of the planning process. Taken all together, the variables and the constraints, both temporal and parameter, form a network of variables linked by constraints, i.e, a *constraint network*. The constraint network is a dynamic entity, as variables and constraints can be added and removed throughout the planning process. The constraint network plays an important role in this approach to planning, since any plan which gives rise to an inconsistent constraint network cannot possibly be extended to a valid plan.

The planner uses *timelines* to represent and reason about the set of possible developments for attributes. For each attribute of each domain object, the planner has exactly one timeline. The reason for utilizing such a specialized construct is that there is a strong relation between tokens that apply to the same attribute of the same object, i.e, the same timeline. Consider any two tokens for the same timeline, each describing a set of valid attribute values for a temporal interval. If the sets of attribute values do not overlap, then the two tokens cannot overlap in time, i.e, one must come before the other. Conversely, if any two tokens necessarily overlap, then they must describe the same interval having the same attribute value. Conceptually, a timeline consists of a sequence of timepoints, each representing a possible transition from one attribute value to another, i.e, the start or end of a token. The interval between any two adjacent timepoints is called a *slot*. During the planning process, a slot will either contain one or more codesignated tokens, or it will be empty.

A set of tokens, along with the associated parameter variable domains, temporal constraints and timelines, describes a *partial plan*. The goal of the planning process is to modify this partial plan, until it is a complete and valid plan. The key observation behind this process is that for any given partial plan, there are only four requirements that can prevent a partial plan from being complete and valid:

1. Parameter variables must be assigned values
2. Tokens must be scheduled onto timelines
3. Configuration constraints must be satisfied
4. Underlying constraint network must be consistent

Any violations of the first requirement can be addressed by selecting a value to assign to each unassigned parameter variable. The second requirement can be enforced by selecting a suitable (not necessarily empty) slot for each uninserted token, and insert the token there. Depending on whether the slot is empty or not, the token will be scheduled between two other tokens or codesignated with a previously scheduled token. The third requirement can be satisfied without any selection criterion. The simplest approach is to instantiate any tokens required to satisfy a configuration constraint, as soon as a token is inserted on a timeline and all parameter domains have been grounded. If the token is later removed from a timeline or the parameter domains are relaxed, then the instantiated tokens are also removed. Finally, if the constraint network is found to be inconsistent, one or more constraints and value assignments can be removed.

Needless to say, the above methods for enforcing the four requirements interact with one another, one fix causing another break. The process of navigating through these operations is called search, and it can be a complex and expensive process. However, in this framework, there are only three relatively simple operations that require decisions to be made, namely:

- Insert a token on a timeline
- Remove token from timeline
- Modify domain of variable, which includes assigning single values

Although having a simple set of operations does not by itself reduce the cost of searching, it does provide a great deal of flexibility in how the search is done. However, the resulting flexibility may lead to significant reductions in search costs, as more effective search techniques can be brought to bear.

4. THE SYSTEM MODULES

One of the key goals of this work is to design and implement a flexible, extendible and portable planning system that can serve as a research framework for further development of autonomous planning and reasoning techniques, while also providing the core for future applications of the Remote Agent Planning technology. The new system is written in C++, to provide structured programming, fast execution and portability. As of May 1999, the redesign is complete, the implementation is almost complete and testing is under way.

The new implementation is based on a careful object-oriented modular design, which allows modules to be easily replaced, improved and tested. Figure 3 shows an overview of the main modules and the relations between them.

The *constraint network manager* is the constraint reasoning module, responsible for handling the dynamic constraint network described above. The main responsibilities are:

- Add and remove variables.
- Add and remove constraints.
- Manage and reason about variable domains.

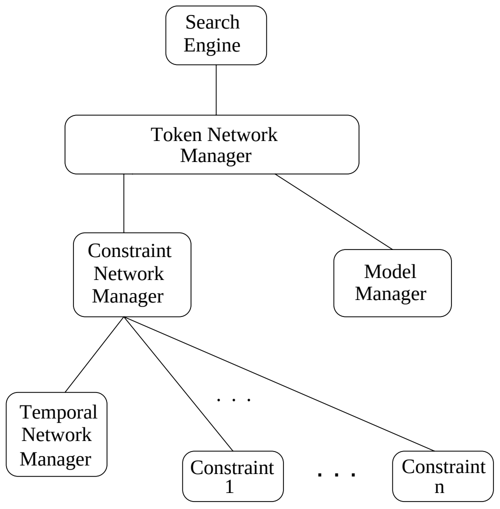

Figure 3: An overview of the key modules in the Next Generation Remote Agent Planner.

- Inform about local and global consistency.
- Provide heuristics for variables and values.

The constraint network manager utilizes the temporal network manager for handling the temporal variables and the temporal constraints that connect them. This is done to allow more efficient algorithms to be applied to the computationally simpler temporal network (Dechter *et al.* 1991). The constraint network also uses external constraint procedure for representing constraints. Such external procedures can represent any constraint, ranging from simple arithmetic equalities to the complicated feasibility evaluations. The design and capabilities of our constraint reasoning framework are discussed further in the next section.

The *model manager* handles all the information relating to the domain model. As a result, it serves a dual role; as the input module responsible for setting up the domain model, and as an information module responsible for providing information about the domain model.

To facilitate the model manager's role as an input module, it has a well-defined input interface that can serve as the single interface for the various different ways in which a model can be specified. As a result, it can be connected to a parser for reading domain descriptions from input files, just as well as it can be connected to a graphical user-interface for building models interactively.

In its role as an information module, the model manager is responsible for effectively responding to queries about the domain model. This includes providing information about the hierarchy of domain classes, the attribute definitions and the predicate definitions. However, most of the work done by the model manager is in providing information about configuration constraints. As the constraints are described by configuration constraint schemata, the model manager can map any given set of attribute values into the applicable configuration constraint instantiation. Furthermore, to facilitate incremental reasoning, it can also determine what changes occur in the applicable configuration constraints, given any two sets of attribute values.

The *token network manager* handles the top-level planning operations, thus providing the interface that the search engine will use. Its main responsibilities are the following:

- Initialize timelines and tokens according to the domain model and the set of goals to be achieved.
- Add/remove temporal constraints between token timepoints.
- Insert and remove tokens from timelines. This includes inserting into empty slots and codesignating with existing tokens.
- Provide access to parameter variables in tokens so that their domains can be modified and assigned values.
- Automatically generate and eliminate tokens in response to applicable and instantiated configuration constraints.
- Determine consistency and validity for the current partial plan.

Finally, on top of the token network manager, there is a *search engine* that controls the planning process. As mentioned above, only a small set of operations is required to modify the partial plan during the planning process. The role of the search engine is to control the application of these operations, with the goal of finding a valid and complete plan.

Recall that the only required operations were the ability to modify a parameter variable domain and the ability to insert and remove tokens from timelines. Any of these operations can be undone by performing another operation from the set. For example, assigning a single value to a variable can be undone by modifying the variable domain to have the set of values it had before. More importantly, the semantics of the operations guarantee that the effect of undoing an operation is is the same as not performing the original operation. This holds regardless of what has been done in between, which is exactly what allows us to utilize non-chronological methods in the search engine.

The added flexibility available to the search engine opens a number of possibilities in making the planning process more efficient. In other domains, various search engines have proven to be effective at solving decision problems such as planning, even in real-world domains. Among the many candidate search techniques that may prove applicable to this planning framework are dependency-directed search (Stallman & Sussman 1977), limited discrepancy search (Harvey 1995), relevance-bounded search (Bayardo Jr. & Miranker 1996), iterative sampling (Langley 1992), heuristic-biased sampling (Bresina 1996) and repair-based search (Minton *et al.* 1990).

5. THE CONSTRAINT REASONING SYSTEM

The Next Generation Remote Agent Planner is based on a redesign of the existing RA planner and thus inherits a number of existing solutions and algorithms. However, a completely new framework has been developed and implemented for doing the constraint reasoning. The new constraint reasoning framework is very general, as it can reason about any set of variables and constraints. At the same time, it is also quite efficient as it combines efficient internal reasoning methods with fast external special-purpose procedural methods.

A constraint network consists of a set of variables, each taking values from a given domain, and a set of constraints connecting the variables. Formally, a constraint is a relation that specifies which combinations of values are allowed for the set of variables in the constraint's scope. However, this is not how constraints are specified in practice, as listing the allowed combinations requires excessive amounts of space. As a result of this, constraints are typically specified using special-purpose constraint descriptions that the constraint reasoning system can understand. In this system, for example, temporal constraints are specified by noting the two variables and the bounds on the distance from one to the other. The problem with this approach is that although it is very efficient and easy to use, it limits the set of constraints to those specifiable in this description language. To solve this problem, without incurring significant efficiency penalties, the Remote Agent constraint network manager can handle external constraint procedures.

A *constraint procedure* is a program that that is applied to a set of variables, the scope of the constraint. The procedure implements a mapping that maps each variable domain to a subset (although not necessarily a strict subset) of that domain. In other words, the procedure reduces the set of possible value assignments for the variables, by eliminating values from the domains. To see how this defines a constraint, let us consider applying the procedure to a set of domains where each domain has only one value, i.e, a variable assignment. The procedure can then either map the set of domains to itself (indicating that this is a valid assignment to the variables) or reduce one or more domains to the empty set (indicating that the given assignment is invalid). A procedure therefore implicitly defines a set of allowed value assignments for the variables in the scope; in other words, it defines a constraint. The only restriction placed on a constraint procedure, in order to make it useful for constraint reasoning, is that it never eliminate any allowed assignments when reducing the domain sets.

The key reasoning task in a dynamic constraint reasoning system is to try to prove the network consistent or inconsistent. This is done by applying a technique called *propagation*, where information about possible and impossible solutions is propagated between variables, through the constraints. In general, correctly determining consistency is NP-complete and will therefore have a worst-case complexity that is exponential in the number of variables. As a result of this, dynamic constraint reasoning is typically done with limited propagation techniques like maintaining arc-consistency.

In its simplest form, *arc-consistency* guarantees that for each value in the domain of a given variable, any single other variable can be assigned some value from its domain, without directly violating a single constraint. Maintaining arc-consistency is therefore the process of eliminating any values that do not satisfy the above condition. This can be accomplished with algorithms that have low-order polynomial complexity. The tradeoff is that inconsistencies may remain undetected, as there is no guarantee that three or more variables can be assigned values without violating a constraint. However, the fact that inconsistencies may remain undetected is not a problem in this planning framework. The reason is that any uninstantiated variables are eventually assigned single values, and in that situation arc-consistency is sufficient to determine the overall consistency correctly.

As in most other dynamic constraint reasoning systems, a propagation algorithm is the core of the constraint network manager. The algorithm we have developed is based on maintaining arc-consistency, but it has been extended so that it can take advantage of other methods that also eliminate values from variable domains. The advantages of this extension are twofold. First, it allows the propagation to directly take advantage of the procedural constraints, which can often eliminate values faster and more effectively than the arc-consistency maintenance. Secondly, the propagation method can be augmented with other efficient propagation algorithms such as the one that performs the propagation within the temporal subnetwork.

The result of all this is not only an efficient framework for performing constraint reasoning, but one that can easily be extended. Constraint procedures can be written separately and simply added to the system, without any modification to the constraint reasoning mechanism. In addition to that, specialized techniques for handling certain parts of the network, e.g, the temporal subnetwork, can be added into the constraint network manager with minimal changes.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an overview of the Next Generation Remote Agent Planner, the next step in the continuing evolution of the RA Planner. The new planning system, with the simplified framework and a modular and flexible design, provides a solid foundation for future applications in autonomous planning for spacecraft, and a framework for further research into the many aspects of autonomous planning for real-world systems.

The development of the Remote Agent planning system is ongoing work, as new challenges arise and better reasoning techniques are developed. This gives us both clear near-term goals and a number of interesting research venues for future work. As of May 1999, the planning framework definition and the modular system design have been completed. The system implementation is close to completion and testing is already underway. Aside from concluding the main system tests, the near-term goals include the development and study of different search engines for driving the planning process. For the longer-term goals, there are too many interesting research questions and application opportunities to list them fully in this paper. However, regardless of which goals are pursued, this new system will provide a solid foundation for both further research into autonomous planning techniques and future applications of the Remote Agent Planner.

REFERENCES

Bayardo Jr., R. J. & D. P. Miranker (1996). A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304.

Bresina, J. (1996). Heuristic-biased stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204.

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

Harvey, W. D. (1995). *Nonsystematic Backtracking Search*. PhD thesis, Stanford University, Stanford, CA.

Langley, P. (1992). Systematic and nonsystematic search strategies. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, pages 145–52. Morgan Kaufmann.

Minton, S., M. D. Johston, A. B. Philips, & P. Laird (1990). Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 17–24.

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

Stallman, R. M. & G. J. Sussman (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–96.