Appears in Springer-Verlag's *Lecture Notes in AI* (Proceedings EKAW-97)

Knowledge Acquisition for the Onboard Planner of an Autonomous Spacecraft

Benjamin D. Smith[1]
Kanna Rajan[2]
Nicola Muscettola[2]

[1] Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive M/S 525-3660
Pasadena, CA 91109-8099
smith@aig.jpl.nasa.gov
[2] NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035
{kanna,mus}@ptolemy.arc.nasa.gov

Abstract. Deep Space One (DS1) will be the first spacecraft to be controlled by an autonomous closed loop system potentially capable of carrying out a complete mission with minimal commanding from Earth. A major component of the autonomous flight software is an onboard planner/scheduler. Based on generative planning and temporal reasoning technologies, the planner/scheduler transforms abstract goals into detailed tasks to be executed within resource and time limits. This paper discusses the knowledge acquisition issues involved in transitioning this novel technology into spacecraft flight software, developing the planner in the context of a large software project and completing the work under a compressed development schedule. Our experience shows that the planning framework used is adequate to address the challenges of DS1 and future autonomous spacecraft systems, and it points to a series of open technological challenges in developing methodologies and tools for knowledge acquisition and validation.

1 Introduction

The future of the space program calls for ambitious missions of exploration and scientific discovery. Searching for life on Mars, Europa and elsewhere in the solar system and beyond will require the solution of several challenging technical and organizational problems. A central one is the implementation of increasingly capable and autonomous control systems to ensure both mission accomplishment and mission safety [17, 6, 15]. Without these systems missions will have to be run with the current, traditional approach. This relies on frequent communication with Earth and teams of human experts guiding step by step a mission through its tasks and analyzing and reacting to the occurrence of malfunctions. The cost and logistics difficulties of this approach, however, are so high that it cannot be reasonably carried over to the expected growth of missions and mission capabilities. Autonomy technology is an answer to these problems.

The Remote Agent (RA) [12, 13] will be the first artificial intelligence-based autonomy architecture to reside in the flight processor of a spacecraft and control it for 6 days without ground intervention. The mission on which RA will fly is Deep Space One (DS1), the first deep-space mission of NASA's New Millennium Project. RA achieves its high level of autonomy by using an architecture with three components: an integrated planning and scheduling system (PS) that generates sequences of actions (plans) from high-level goals, a intelligent executive (EXEC) that carries out those actions and can respond to execution time anomalies, and a Model-based Identification and Recovery system (MIR) that identifies faults and suggests repair strategies. Each module covers a different function in the architecture and uses a different computational approach. One characteristic however is common to all of them: the reliance on models of the domain that are largely independent from the task to be fulfilled. These models allow the module to rely on a much deeper understanding of the structural characteristics of the domain than possible with classical rule-based approaches, facilitating model analysis and model reuse.

This paper discusses the knowledge acquisition process used for models and heuristics of the planning and scheduling system (PS) of DS1. We started the process with an approach to planning knowledge representation [10] that had been demonstrated in a rapid-prototype effort [12]. With DS1 we had to face additional challenges due to having to develop PS in the context of the development of the full flight software, to the additional complexity of the domain, to the compressed schedule for development and to the risk-management requirements. Also, architectural solutions internal to RA had to be enhanced due both to the increase in capabilities that were needed to control a real spacecraft and to the need to provide sounder software engineering approaches. We will describe how the knowledge acquisition process was carried out and the strengths and weaknesses we found in our current approach.

Section 2 deals with the Remote Agent software architecture highlighting the details of the planner and it's plan representation. Section 3 deals with issues in knowledge acquisition including references to the spiral development process, model acquisition and interfaces to external experts. Section 4 deals with the open issues as a result of the development process including the need for validation and debugging tools. Conclusions appear in Section 5.

2 The Remote Agent and Planner Architectures

The RA architecture consists of four distinct components (Figure 1), the *Planner/Scheduler*, the *Mission Manager* [11], the *Smart Executive* (EXEC) [14] and the *Mode Identification and Recovery* (MIR) system [16, 17] .

The execution of plans by the EXEC is achieved by interaction with a Mode Identification system and a lower level real time monitoring and control component. MIR provides the EXEC with a level of abstraction to reason about the

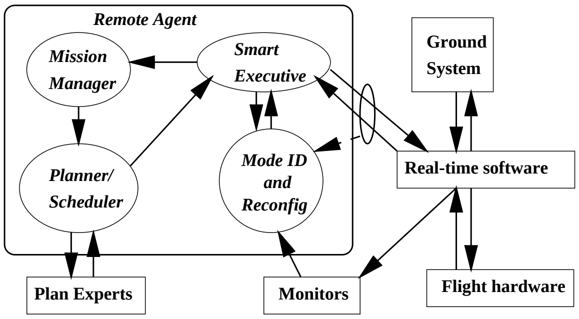

Fig. 1. RA architecture

state of the various devices it commands. The *monitoring* layer takes the raw sensor data and discretizes it to the level of abstraction needed by MIR. Finally, the *control and real-time system* layer takes commands from the executive and provides the actual control of the low level state of the spacecraft. It is responsible for providing the low level sensor data stream to the monitors. Details of the Remote Agent architecture can be found in [12, 13].

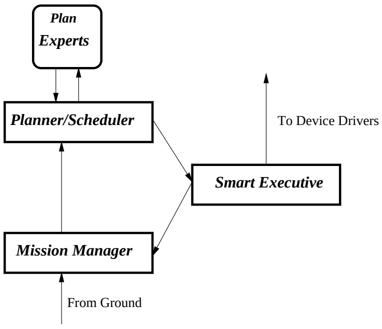

Fig. 2. Planner/Scheduler Architecture

The planner/scheduler (PS) generates a detailed plan of action from a handful of high-level goals, based on knowledge of the spacecraft contained in a domain model. The model describes the set of actions, how goals decompose into actions, the constraints among actions, and resource utilization by the actions. The planning engine searches the space of possible plans for one that satisfies the constraints and achieves the goals. The action definitions determine the space of plans. The constraints determine which of these plans are legal, and heavily prune the search space. The heuristics guide the search in order to increase the number of plans that can be found within the time allocated for planning.

Figure 2 describes the overall view of the Remote Agent Planner/Scheduler.

The Mission Manager (MM) contains the long term mission plan with goals for the entire mission. Ground operators can interact with the MM to modify the plan by adding, removing or editing goals in the mission plan. The MM also provides the EXEC with an interface to the Planner. When the EXEC requests a new plan, MM selects a new set of goals from the mission profile, based on internal way points. It combines this with the initial state provided by the EXEC and generates a partial plan for the planner. When the EXEC has completed execution of this plan the cycle is completed when it sends a new request to the MM for the next planning horizon. For the RA experiment the plan horizon will consist of two segments each three days long.

2.1 Knowledge Representation of the Planner

The knowledge representation of the planner is distributed among the domain models, the planner heuristics, the mission profile and the plan experts. The domain models encode the behavioral and operational constraints imposed on the spacecraft by the mission and the hardware. The heuristics guide the planner search to decrease the computational resources needed to find a plan and to increase plan quality. The mission profile encodes the long term goals and mission requirements as determined by the ground controllers and mission designers, and resides in the Mission Managers temporal database. Finally the plan experts are special-purpose software modules, written and maintained by other teams, with which the planner interacts to obtain knowledge that cannot be easily encoded in the plan model.

Model Representation. The PS uses a hybrid planning/scheduling representation that models continuous processes on parallel timelines to describe actions, states and resource allocations. PS provides also for temporal and parametric flexibility and uses planning experts.

Plans consist of several parallel *timelines*, each of which consists of a sequence of *tokens*. A timeline describes the evolution of a spacecraft state over time, and the tokens describe those states. For example, consider one timeline that describes the main engine. If the plan is to start in standby, fire up the engine, and return to standby, the timeline would have one token for each of those processes. Each token has a start time, and end time, and a duration. Each token can have zero or more arguments (e.g., the thrust level at which to fire the engine).

The plan model consists of definitions for all the timelines, definitions for all the tokens that can appear on those timelines, and a set of temporal constraints that must hold among the tokens in a valid plan. The planner model is described in a domain description language (DDL), and is represented as part of the planner's data base also called the Plan DB.

Temporal constraints are specified in DDL by *compatibilities*. A compatibility consists of a *master token* and a boolean expression of temporal relations that must hold between the master token and *target tokens*. An example is shown in Figure 3.

```
(Define_Compatibility
  ;; compats on SEP_Thrusting
  (SEP_Thrusting ?heading ?level ?duration)
  :compatibility_spec
  (AND (equal (DELTA MULTIPLE (Power) (+ 2416 Used)))
       (contained_by (Constant_Pointing ?heading))
       (met_by (SEP_Standby))
       (meets  (SEP_Standby))))

(Define_Compatibility
  ;; Transitional Pointing
  (Transitional_Pointing ?from ?to ?legal)
  :parameter_functions
    (?_duration_ <- APE_Slew_Duration (?from ?to ?_start_time_))
    (?_legal_    <- APE_Slew_Legality (?from ?to ?_start_time_))
  :compatibility_spec
  (AND (met_by (Constant_Pointing ?from))
       (meets  (Constant_Pointing ?to))))

(Define_Compatibility
  ;; Constant Pointing
  (Constant_Pointing ?target)
  :compatibility_spec
  (AND (met_by (Transitional_Pointing * ?target LEGAL))
       (meets  (Constant_Pointing     ?target * LEGAL))))
```

Fig. 3. An example of a compatibility constraint in the planner model.

The first compatibility says that the master token, `SEP_THRUSTING` (when the Solar Electric Propulsion engine is producing thrust), must be immediately preceded and followed by a standby token, temporally contained by a constant pointing token, and requires 2416 Watts of power. Constant pointing implies that the spacecraft is in a steady state aiming its camera towards a fixed target in space. Transitional pointings turn the spacecraft. The SEP standby state indicates that the engine is not thrusting but has not been completely shut off. A plan fragment based on these compatibilities is shown in Figure 4.

Heuristics. Heuristics guide every choice point of the planners search. On each iteration of the search, the planner chooses an unresolved compatibility constraint and a way to resolve it: by constraining an existing token to satisfy the constraint, adding a new token that satisfies it, or assuming that it will be satisfied by some token in the next horizon. There are other decisions as well, such as grounding under-constrained argument values.

For example, the heuristic in Figure 5 prevents backtracking due to cycling. The attitude timeline alternates between constant pointing tokens and transi-

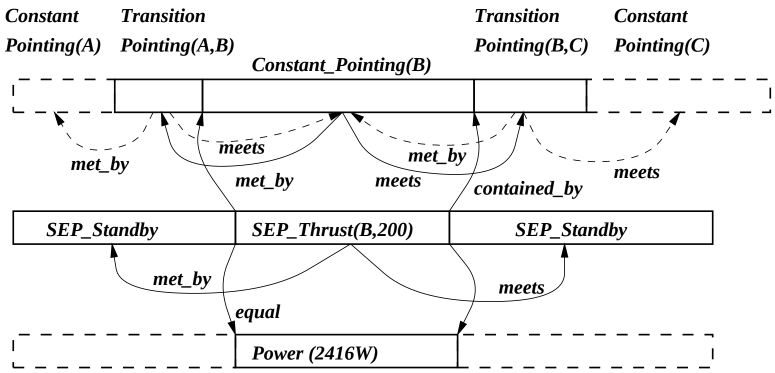

Fig. 4. Plan Fragment

```
Method Connect
   ;; connect only, defer and add are disallowed.
   master { Transitional_Pointing; }
   target { Constant_Pointing; }
```

Fig. 5. Cycle Avoidance Heuristics

tional pointing tokens. Once a constant pointing token is added, it has compatibilities that could add transitional pointing tokens, which can in turn add constant pointing tokens add infinitum. The last heuristic breaks the cycle by saying that a transitional pointing token can *connect* to an existing constant pointing token, but can never *add* one.

Mission Profile. The goals for the entire mission are stored in an on-board file called the *mission profile*, which is managed by the Mission Manager. The profile captures mission operations knowledge, such as when the communications passes are scheduled, how much fuel is allocated for each segment of the mission, when various mission phases start and stop, and so on. The profile also serves as the primary interface with the ground controllers. The ground team commands the spacecraft at a high level by changing or adding goals to the profile.

Plan Experts. A large software project like the DS1, requires the contribution of several teams with specialized knowledge. *Planning Experts* are programs developed and maintained by other teams. They coordinate with the planner but which are not strictly part of its domain representation.

A prime example is the Attitude control Planning Expert (APE), which answers queries about how long the turn will take and whether a given turn violates pointing constraints (e.g., will the turn expose the camera to a damaging bright radiation source). How violation constraints are calculated is completely opaque to the planner. As a result, separating the plan experts from the planning model simplifies the knowledge acquisition and software maintenance process. Quite often, due to the specificity of these modules, the code is also reusable across missions. For instance, much of the code for attitude constraint violation in APE came from NASA/ESA's Cassini mission [5].

There are two kinds of plan experts. The first kind answers questions about constraints. APE is of this variety. The second kind of plan expert generates goals for the planner to achieve. These on-board goal generators allow the spacecraft to make autonomous decisions, within certain parameters, based on local information. The prime example on DS1 is the on-board navigator, which provides goals on trajectory related maneuvers and goals for images of nearby celestial bodies from which NAV can determine the spacecraft position.

The planner asks the goal generators for goals when the planner is ready for them. The goal generators have no visibility into the plan, other than whatever information provided in the request. When the goals are returned, the planner decides how they will be achieved in the plan, or whether they are achieved at all. If the plan is over-constrained, goals can be rejected based on a global prioritization scheme.

3 The Knowledge Acquisition Process

Traditionally flight software for a spacecraft consists only of low level device drivers, attitude control system and simple sequence execution capabilities. Commanding done from ground allows the operational and mission constraints to be designed and implemented at a later time, sometimes even after launch. With on-board autonomy, the design process must take a more comprehensive view to the full mission life cycle including from the *very beginning* the modes, operations and expected behaviors of the spacecraft in the domain models. To accomplish this we used a spiral development model [1].

In the following sections we discuss the knowledge acquisition process and methodology for the planner and the resulting problems and issues they raised.

3.1 The Spiral Development Process

In spiral development [1], functionality is added incrementally in distinct software releases. This allows base functionality to be understood and developed before moving to more complex functionality. Processes and standards are also refined in each spiral. At the end of each development cycle, project teams meet to discuss the obstacles they encountered and the lessons they learned. The DS1 spiral process is discussed further in [8].

At the beginning of each spiral, the mission engineers created a baseline scenario that would exercise the new functionality for that spiral while still requiring the old functionality. The hardware management team (HWMT) then arranged several days of knowledge acquisition meetings with the hardware developers, who would detail the software requirements for their hardware to work correctly.

Each of the modeling and software development teams sent representatives to these meetings. The hardware developers presented the baseline behavior for the upcoming spiral, and the modelers asked questions to elicit further details. Since each component of the RA models the hardware at a different level, having representatives from each team was particularly helpful in identifying interaction issues across the different levels.

The DS1 Spiral releases were designated R1 through R3. To give the reader the scope of development that took place, we show the evolution of the planner model sizes for each revision in Tables 1, 2 and 3.

From the PS perspective each revision in the spiral development model involved successively sophisticated constraint modeling of the spacecraft. In the first revision the model only dealt with simple turns and picture taking for navigation images; more complex issues such as power, thermal modeling were ignored. In the next revision the model included the modeling the SEP engines and obtaining more detailed trajectory information from the navigation expert. The third spiral release added power management, advanced turns, and comet fly-by related activities.

In each revision of the Spiral development approximately eight weeks were needed for knowledge capture and another eight weeks for model development and tuning of the planner search.

Subsystem	R1	R2	R3
Mission events	0	1	3
Power	0	0	2
Ion Propulsion	1	5	5
Attitude control	3	4	4
Communications	0	1	2
MICAS	1	1	6
Beacon experiment	0	0	2
RCS system	1	1	3
Navigation	3	3	4
Planner/scheduler	1	1	1
Total	10	17	32

Table 1. Number of timelines changed in the model for each development release.

3.2 Model Acquisition

Model aquisition in each cycle started with the cognizant system engineer specifying the baseline functionality to be covered, layered on top of designs of subsystems already implemented. Each team then developed a specification called a *Problem Statement* which described how that functionality would be achieved.

Subsystem	R1	R2				R3			
	Tot	Add	Mod	Del	Tot	Add	Mod	Del	Tot
Mission events	0	1			1	5			6
Power	0				0	3			3
Ion Propulsion	1	11	1		12	1	3		13
Attitude control	4	4			8	6	2	1	14
Communications	0	3			3	2	2		5
MICAS	3	5			8	14			22
Beacon experiment	0				0	4			4
RCS system	1				1	4	1		5
Navigation	6		2		6	3			9
Planner/scheduler	2				2				2
Total	17				41				69

Table 2. Number of token modifications to the model for each development release.

Subsystem	R1	R2	R3
Mission events	0	2	4
Power	0	0	0
Ion Propulsion	3	11	14
Attitude control	2	11	16
Communications	0	3	7
MICAS	3	3	20
Beacon experiment	0	0	4
RCS system	0	0	2
Navigation	4	3	7
Planner/scheduler	2	2	2
Total	16	36	76

Table 3. Number of compatibilities changed in the model for each development release.

For the planner, this described changes to the planning model and engine and described changes to interfaces with the EXEC and plan experts. Teams with interfaces with the planner (especially the EXEC) would comment and propose design changes and any additional requirements. After a few iterations of this process, the modeler would update the *Token Dictionary.* The token dictionary details the syntax and semantics of each token type on all the timelines and forms the primary document for all negotiated informal interfaces with the EXEC.

We used an informal elicitation methodology to acquire the models described in the problem statements. The elicitation began with a standard list of questions about how the subsystem operated and what constraints or interactions it had with other subsystems. These would then lead to more detailed questions. The captured knowledge was compiled in a semi-formal document and approved by the engineers in a separate session. There was no formal methodology to ensure coverage other than the constraints of the baseline scenario. Nonetheless, this was adequate to build the plan models and successfully complete the scenarios for each spiral release. In retrospect, an aquisition methodology that resulted in a formal specification with guaranteed coverage would have been useful for rigorously validating the model (see Section 4.3).

Issues in Domain Modeling. In modeling for the DS1 mission, we discovered that a relatively large number of modeling tasks were easy to do, given the syntax and semantics of DDL. In a couple of cases we had to introduce auxiliary timelines to support the planner's reasoning process.

For instance it was necessary to schedule the correct amount of `SEP thrust` within a planning horizon. Simply put, this would require computing the duration of each `SEP thrust` token and performing a summation over the planning horizon. To do so however, a somewhat circular approach was needed by defining a new timeline which would use variable codesignation and propagate accumulated thrust values based on temporal durations of the `SEP thrust` tokens; something not altogether intuitive. Figure 6 illustrates this situation with the `Max Thrust` duration specifying the thrust duration *needed* in the horizon; `SEP thrust` tokens actually *performing* the thrusting action and `Accumulated thrust` tokens showing a running *count* of the thrust.

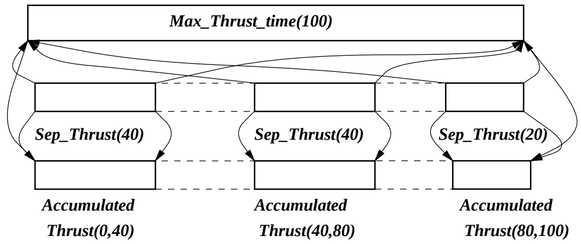

Fig. 6. A plan fragment implementing thrust accumulation within a plan horizon

As mentioned earlier, with each iteration of the development cycle the planner models were made consistently more realistic. Knowledge acquisition from each spiral cycle effectively ended up affecting the planner's domain model and it's heuristics. While syntactic modifications and its semantic interpretation were the primary drivers of model development (especially if interface requirements at the token level were to be negotiated with the EXEC (see Section 3.4)) we discovered that significant development time was spent aligning the heuristics with the model changes so that planner convergence was possible.

Heuristics. Because of the tight coupling of the domain model to the heuristics, changes to the model almost always require corresponding changes to the heuristics. This makes it difficult to introduce incremental changes to the model. Normally, a family of timelines corresponding to a new device or capability can be added with minimal impact on other timelines. Most of the constraints are among the timelines in the family, with a handful of constraints to external resources such as power or spacecraft attitude. However, the new timelines change the optimal search strategy, and this requires the heuristics to be re-tuned.

3.3 Unmodeled Activities

Sometimes the ground controllers want to execute unusual maneuvers that are not modeled the planner. For example, they may want to execute a high-speed turn (normally disallowed) in order to jar loose a stuck solar panel. The model must provide a way for the ground controllers to execute contingency maneuvers such as this without uploading a new model. Changing the model maybe fine for permanent patches, but the time and cost needed to develop and test the patch makes them impractical for one-time emergency maneuvers.

In support of this requirement the model provides a `special activity` token that can stand in for any activity the ground wants to execute but is not otherwise supported by the planning model. The ground controllers insert the token where they want it in the mission profile. It can be scheduled for a specific time, or scheduled relative to other events. The activities performed by the token are specified in a file of time-tagged low-level commands that the EXEC executes at the beginning of the token.

Since the actions executed in the special activity token are not modeled by the planner, it is possible that they could conflict with planned activities. For example, the plan could be trying to hold the spacecraft still in order to take an image while the special activity token is executing a high speed turn. To avoid such conflicts, constraints can be specified between the special activity token and other tokens in the plan. In this case, attitude dependent activities would be disallowed while the special activity token was active. These constraints can be specified in the mission profile.

3.4 Interfaces

The interfaces between the planner and other parts of the flight software impact the knowledge acquisition and representation. The planner has two main interfaces: interfaces with *plan experts*, and interfaces with the *Smart Executive* component of the RA.

Plan Expert Interfaces. Negotiating the plan expert interfaces was among the easiest of the knowledge acquisition tasks. This is largely due to the opaqueness of the plan experts to the planner and vice versa. The bulk of the knowledge acquisition was in the very first meeting, where the focus was understanding how the plan expert worked and explaining planner concepts to the plan expert developers. In the case of APE, the planning team needed to understand how to specify a turn, and what information was needed for APE to compute a turn. The details of how turns are computed were irrelevant.

Once this initial knowledge acquisition was completed, subsequent interface negotiations were completed in a matter of hours, usually by phone or email. The interfaces were formally defined as C structures that specified the information passing from the planner to the plan expert and back. These were captured in an interface control document, and in an executable interface specification language.

In some cases, the planner used assumptions about the inner workings of the plan expert to improve efficiency. For example, the legality and duration of a turn changes slowly and continuously over time. This allows a turn to be moved a couple hours ahead or back in the plan as needed without affecting its duration or legality. The planner model and heuristics exploited this knowledge to simplify the design and speed up the search. Assumptions of this sort were rare, and captured explicitly in the interface control documents.

EXEC Interfaces. The interfaces between the planner and the Smart Executive (EXEC) are embodied by the timeline and token definitions included in the planner's model.

In order for the EXEC to execute plans, it must know what tokens can appear in the plan and how to expand those tokens into detailed commands to the real-time flight software. This creates a very strong coupling between EXEC and the planner. All of the timelines, tokens, and their semantics were negotiated at the beginning of each spiral before any implementation took place. However, if the need for another token was discovered during development, or some token needed another argument, or the semantics were wrong, then the EXEC and PS had to change their implementations accordingly. Because the tokens are such a major part of the model implementation, changes of this sort occurred in every development spiral despite strong efforts to prevent them.

Several solutions to this interface issue were considered for DS1. One successful approach was to use *information hiding* to create private token arguments. Additional arguments are often needed to hold values derived from other arguments, or to propagate values from other tokens. The need for these arguments often goes unnoticed until development begins in earnest. Private arguments are seen by the planner, but are do not appear in the plan. This allowed modelers to add arguments for bookkeeping and propagation purposes without impacting the EXEC. This capability was introduced at the end of the R2 spiral, and used with great success in R3.

Interface Management Process. To ensure disconnects were kept to a minimum, another requirement added by the project during the design phase of each revision, was the development of *Problem Statements*, with details of each modules' design, interfaces and assumptions for that revision cycle. The planner in addition also had a token dictionary with the negotiated token level interfaces with the EXEC. With the EXEC with which the planner representation was tightly coupled, any agreements and assumptions in the planner's model were accurately document and easily accessible via a world wide web (WWW) interface to the dictionary and disconnects caught early on. In order to avoid disconnects with respect to the hardware specifications, especially as hardware delivery quantified the performance, the HWMT was the central point of contact for disseminating information.

The project also made sure that after the interface parties and the design phase for each cycle, but *before* the various teams started actually developing code, a *concept review* would take place. Each team would publish a short document detailing their design choices and the assumptions made, especially towards generating the scenario in the current cycle and the interface requirements. Any disconnects found would require the project to follow through with the team in question to ensure the new design actually covered the complete scenario.

3.5 Distributed Development

Because of geographically distributed teams, design documents and interface agreements were exchanged primarily via a WWW interface with auto-posting features as mentioned in [2]. The use of the Internet was decisive in successfully collaborating over remote sites especially when exchanging device level knowledge. For instance, the HWMT would be able to post power budget allocations over different devices which could be abstracted, parsed and built into a table for lookup during model generation time. This would ensure that the planner models were consistent in their power budgets with those of the system engineering and actually viable for generating robust plans.

Additionally, for short design and concept reviews, a "meet-me" telephonic system was in place with people calling into a central number while accessing the web and viewing the same document simultaneously. This greatly helped in cutting down the time, effort and expense of commuting to a central site. Note also that a revision control system [3] was in place for all the source code.

4 Open Issues

The DS1 project presented several challenges in knowledge acquisition, representation, and validation. The DS1 planner proved capable of addressing these issues, at least to the extent needed to satisfy the requirements of DS1. However, there are a number of issues that must be resolved before this technology can be used on a risk-intolerant science mission by spacecraft engineers with minimal support from the planner development team.

4.1 Acquiring Heuristics is Difficult

Good heuristics are needed to make the planner search computationally tractable. Heuristics tell the planner what decisions are most likely to be best at each choice point in the planner search algorithm, thereby reducing the search. Developing a good set of heuristics for the DS1 planner is currently very difficult, both because it requires an intimate knowledge of how the planner search algorithm interacts with the model, and because the planner requires exceptionally good heuristics to achieve computational tractability. The DS1 model developers had this experience and so were able to develop good heuristics, but these obstacles must be overcome before spacecraft engineers can be expected to develop heuristics on their own.

One solution is to provide tools that derive heuristics automatically. Such tools have been discussed in the machine learning and planning literature. Two of the more promising approaches are to derive heuristics automatically through a static analysis of the plan model [4] or to learn them by watching the behavior of the planner over several runs on a given model [9]. Unfortunately, the DS1 planner requires exceptionally good heuristics to achieve tractability, and these methods generally do not produce heuristics of that caliber. The sensitivity of the planner to the heuristic must be reduced before automatic heuristic acquisition can be feasible.

4.2 Development and Debugging Tools are Needed

Modeling could be made considerably easier with even a few simple tools. Although there was insufficient time to develop them for DS1, our experience with developing and debugging models suggested a number of desirable features. Developing tools along these lines is one of our near term goals.

Plan Visualization Tools. One problem with the current system is that it is very difficult to understand what the planner is doing, despite copious output. This makes it difficult to isolate the decisions that lead to bugs in the plan, or prevent the planner from finding any plan at all. A visualization tool would help modelers to track the planners behavior, as well as making it easier for new users to understand how the planner works.

Deactivating Timelines. When debugging, the modeler often suspects the bug is within a small family of timelines. But as the model gets more complex, it becomes difficult to focus on the behavior of those timelines. A simple way to address this problem is to disable irrelevant timelines. The planner ignores the timelines and all compatibilities associated with them. This facility is rather easy to add, though there was insufficient time to implement given the compressed DS1 schedule.

Model Visualization Tools. As the model gets larger, it becomes harder to keep in mind all the constraints among the parts of the model. A model visualization tool that displayed a graphic view of the model (or a subset) and the constraints would help the modeler view this information as a whole.

4.3 Large Scale Validation of the Remote Agent Planner

Before any spacecraft is launched, its flight software must be throughly tested and validated. The same is true for autonomous flight software. However, the validation methods used for traditional software are not generally applicable to autonomous software. New methods must be developed.

The planner can generate several thousand different plans, depending on the mission goals, the spacecraft state, goals generated on-board by plan experts and variations on the model parameters. To fully validate the planner, one must be sure that it will generate a correct plan for every one of those possible situations, and that the plan can be executed correctly by the EXEC.

An alternative to generating and testing several thousand plans is to use formal methods to validate the models and to verify that the planner produces plans that are consistent with the model. One approach is to capture the flight rules and other requirements formally as constraints, and ensure that the model is consistent with all of them, and that no constraints have been missed. A related possibility is to convert the models into a human-readable form and have them approved by cognizant system engineers (domain experts).

Tools for automatically generating and validating plans can greatly reduce this cost. One of the tools we considered for DS1 but did not have time to implement was a constraint checker that tested whether the plan satisfied certain correctness constraints. These include the constraints in the model, plus additional constraints derived from flight rules and other operational constraints.

5 Conclusion

DS1 will be the first deep-space spacecraft under autonomous control. The complexity of this domain raised a number of important knowledge acquisition and representation issues, some of which we were able to address and some of which remain open. Issues were also raised by the fast-paced spiral development cycle, the embedding of the planner within the autonomy architecture, and the risk-management requirements of the space flight domain.

These issues are not unique to DS1, and are likely to occur on other projects that require autonomous control of a complex environment. As the role of autonomy increases in space exploration and other areas, so will the importance of finding good solutions to these issues.

6 Acknowledgments

This paper describes work partially performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration. This work would not have been possible without the extraordinary effort and dedication of the rest of the Remote Agent Planning Team: Steve Chien, Charles Fry, Sunil Mohan, Paul Morris, Gregg Rabideau, and David Yan.

References

1. Barry Boehm. A Spiral Model of Software Development and Enhancement. *Computer*, pages 61–72, May 1988.
2. Michael Compton, Helen Stewart, Vinod Baya, Martha Del Alto, Bob Kanefsky, and Jason Vincent. Electronic collaboration for the New Millenium: Internet-based Tools and Techniques for Sharing Information. In *http://ic-www.arc.nasa.gov/ic/projects/nmp-doc/nmp-doc-pres.pdf*, 1997.
3. Per Cederqvist et.al. Concurrent Versions System. In *http://www.loria.fr/ molli/cvs-index.html*, 1996.
4. Oren Etzioni. Acquiring Search Control Knowledge via Static Analysis. *Artificial Intelligence*, 62, 1993.
5. G.M.Brown, D.Bernard, and R.Rasmussen. Attitude and Articulation Control for the Cassini Spacecraft. a fault tolerance overview. In *14th AIAA/IEEE Digital Avionics Conference*, 1995.
6. Barbara Hayes-Roth. An Architecture for Adaptive Intelligent Systems. *Artificial Intelligence*, 72, 1995.
7. IEEE. *Proceedings of the IEEE Aerospace Conference*, Snowmass, CO, 1997.
8. Sanford Krasner and Douglas E. Bernard. Integrating Autonomy Technologies into an Embedded Spacecraft System—Flight Software System Engineering for New Millennium. In *Proceedings of the IEEE Aerospace Conference* [7].
9. Steven Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1), 1996.
10. Nicola Muscettola. HSTS: Integrating planning and scheduling. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.
11. Nicola Muscettola, Ben Smith, Charles Fry, Steve Chien, Kanna Rajan, Gregg Rabideau, and David Yan. On-Board Planning for New Millennium Deep Space One Autonomy. In *Proceedings of the IEEE Aerospace Conference* [7].
12. Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. A Remote Agent Prototype for Spacecraft Autonomy. In *Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation*, 1996.
13. Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An Autonomous Spacecraft Agent Prototype. In *Proceedings of the First International Conference on Autonomous Agents.* ACM Press, 1997.
14. Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Plan Execution for Autonomous Spacecraft. In Louise Pryor, editor, *Procs. of the AAAI Fall Symposium on Plan Execution.* AAAI Press, 1996.
15. M. Tambe, W. Lewis Johnson, R. M. Jones, F. Koss, J. E. Laird, Paul S. Rosenbloom, and K. Schwamb. Intelligent Agents for Interactive Simulation Environments. *AI Magazine*, 16(1):15–39, Spring 1995.
16. Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Procs. of AAAI-96*, pages 971–978, Cambridge, Mass., 1996. AAAI Press.
17. Brian C. Williams and P. Pandurang Nayak. Immobile Robots, AI in the New Millennium. *AI Magazine*, Fall, 1996.