Remote Agent: An Autonomous Control System for the New Millennium

Kanna Rajan[1] **Douglas Bernard**[4] **Gregory Dorais** [1] **Edward Gamble** [4] **Bob Kanefsky** [3] **James Kurien** [2]
William Millar [3] **Nicola Muscettola** [2] **Pandurang Nayak** [1] **Nicolas Rouquette** [4]
Benjamin Smith [4] **William Taylor** [3] **Yu-wen Tung**[4]
kanna@ptolemy.arc.nasa.gov

Abstract. On May 17th 1999, the Remote Agent (RA) became the first Artificial Intelligence based closed loop autonomous control system to take control of a spacecraft. The RA commanded NASA's New Millennium Deep Space One spacecraft when it was 65 million miles away from earth. For a period of one week this system commanded DS1's Ion Propulsion System, its camera, its attitude control and navigation systems. A primary goal of this experiment was to provide an on-board demonstration of spacecraft autonomy. This demonstration included both nominal operations with goal-oriented commanding and closed-loop plan execution, and fault protection capabilities with failure diagnosis and recovery, on-board replanning following unrecoverable failures, and system-level fault protection.

This paper describes the Remote Agent Experiment and the model based approaches to Planning and Scheduling, Plan Execution and Fault Diagnosis and Recovery technologies developed at NASA Ames Research Center and the Jet Propulsion Laboratory.

Keywords: closed loop control, constraint-based planning, scheduling, temporal networks, spacecraft autonomy

1 Introduction

May 1999, represents a milestone in the history of the development of spacecraft autonomy. In two separate experiments, the Remote Agent, an AI software system, was given control of an operational NASA spacecraft and demonstrated the ability to respond to high level goals by generating and executing plans on-board the spacecraft, all the time under the watchful eye of model-based fault diagnosis and recovery software.

Current spacecraft control technology relies heavily on a relatively large and skilled mission operations team that generates detailed time-ordered sequences of commands or macros to step the spacecraft through each desired activity. Each sequence is carefully constructed in such a way as to ensure that all known operational constraints are satisfied and the autonomy of the spacecraft is limited. The costs associated with such a process increase with the complexity of the mission. Further, NASA has ambitious plans to fly spacecraft constellations for near earth (for weather forecasting and in-situ measurements of the atmosphere's characteristics) and deep space missions (for interferometry for planet finding and robust mission execution). Yet another motivation for closing the loop on-board, is to save valuable time in the use of NASA's Deep Space Network (DSN) of antennas which have to be in constant contact with spacecraft at distances where the round-trip delay time can be in the order of hours. The DSN is an oversubscribed resource for commanding spacecraft and targeting on-board commanding is a justifiable way to reduce mission operation costs. The cost/benefits therefore, of having an autonomous system control substantial portions of the routine commanding of such missions, can then be translated into more missions that could be flown with the *same* pool of mission operators.

The Remote Agent (RA) approach to spacecraft commanding and control puts more "smarts" on the spacecraft. In the RA approach, the operational rules and constraints are encoded in the flight software and the software may be considered to be an autonomous "remote agent" of the spacecraft operators in the sense that the operators rely on the agent to achieve particular goals. The operators do not know the exact conditions on the spacecraft, so they do not tell the agent exactly what to do at each instant of time. They do, however, tell the agent exactly which goals to achieve in a specified period of time.

Three separate Artificial Intelligence technologies are integrated to form the RA: an on-board planner-scheduler, a robust multi-threaded executive, and a model-based fault diagnosis and recovery system [12, 9]. This architectural approach was flown on the NASA's New Millennium Program Deep Space One (DS1) spacecraft as an experiment.

The DS1 Remote Agent Experiment (RAX) had multiple objectives [2]. A primary objective of the experiment was to provide an on-board demonstration of spacecraft autonomy. This demonstration included nominal operations with goal-oriented commanding and closed-loop plan execution, a demonstration of fault protection capabilities with failure diagnosis and recovery, on-board replanning following unrecoverable failures and finally, system-level fault protection. These capabilities were demonstrated using in-flight scenarios that included ground commanding and simulated failures.

Other equally important, and complementary, goals of the experiment were to decrease the risk (both real and perceived) in deploying RAs on future missions and to familiarize the spacecraft engineering community with the RA approach to software integration and spacecraft command and control. These goals were achieved by a three-pronged approach. First, a successful on-board demonstration required integration of the RA with the spacecraft flight software. This integration provided valuable information on required interfaces and performance characteristics, and alleviated the risk of carrying out such integration on future missions. Second, a perceived risk of deploying an RA is related to its ability to synthesize new untested

[1] RIACS, NASA Ames Research Center, Moffett Field, CA 94035
[2] MS-269-2 NASA Ames Research Center, Moffett Field, CA 94035
[3] QSS Inc., NASA Ames Research Center, Moffett Field, CA 94035
[4] Jet Propulsion Laboratory, Pasadena, CA 91109

sequences in response to unexpected situations. We addressed this risk by demonstrating a layered testing methodology that serves to build confidence in the sequences synthesized by the RA in a variety of nominal and off-nominal situations. Third, the experiment was operated with close cooperation between RA team members and DS1 ground operators. This served to familiarize the ground operations community with benefits and costs of operating a spacecraft equipped with an RA.

The RAX was successfully executed on-board DS1 in the week of May 17–21, 1999 during the ballistic cruise phase of the mission. Additional details on the experiment itself can be found in [3], [11] and [15].

2 The Remote Agent Architecture

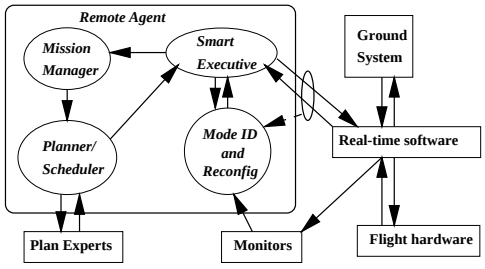

Figure 1. The Remote Agent Architecture

The Remote Agent consists of general-purpose *reasoning engines*, and mission-specific *domain models*. The engines make decisions and command the spacecraft based on the knowledge in the models. This section describes the details of the reasoning engines and how they interact. The architecture of the RA is given in Figure 1.

The overall system had to operate under stringent performance and resource requirements. The DS 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. Moreover, only 45% peak use of the CPU was available for RAX, the rest being used for the real-time flight software. The response times of MIR and EXEC was no greater than 5 seconds. Planning being an expensive activity, the PS and EXEC loop was closed in the order of hours, usually within two. Planning itself was invoked a batch process.

Each of the critical modules in this system is described in greater detail in the following sections: 2.1 (PS), 2.2 (MIR) and 2.3 (EXEC).

2.1 The Planner/Scheduler

The Planner/Scheduler (PS) provides the core of the high-level commanding capability of RA. Given an initial, incomplete plan containing the initial spacecraft state and goals, PS generates a set of synchronized high-level activities that, once executed, will achieve the goals. In the spacecraft domain, planning and scheduling aspects of the problem need to be tightly integrated. The planner needs to recursively select and schedule appropriate activities to achieve mission goals and any other subgoals generated by these activities. It also needs to synchronize activities and allocate global resources over time (e.g., power and data storage capacity). Subgoals may also be generated due to limited availability of resources over time. For example, it may be preferable to keep scientific instruments on as long as possible (to maximize the amount of science gathered). However limited power availability may force a temporary instrument shutdown when other more mission-critical subsystems need to be functioning. In this case the allocation of power to critical subsystems (the main result of a scheduling step) generates the subgoal "instrument must be off" (which requires the application of a planning step). PS is able to tune the order in which decisions are made to the characteristics of the domain by considering the consequences of action planning and resource scheduling simultaneously. This is a significant difference with respect to classical approaches both in Artificial Intelligence and Operations Research, where action planning and resource scheduling are typically addressed in two sequential problem-solving stages, often by distinct software systems [16].

Another important distinction between PS and other classical approaches to planning is that besides activities, the planner also schedules the occurrence of states and conditions. Such states and conditions may need to be monitored to ensure that, for example, the spacecraft is vibrationally quiet when high stability pointing is required. These states can also consume resources and have finite durations and, therefore, have very similar characteristics to other activities in the plan. PS explicitly acknowledges this similarity by using a unifying conceptual primitive, the *token*, to represent both actions and states that occur over time intervals of finite extension. PS consists of a heuristic search engine that deals with incomplete or partial plans. Since the plans explicitly represent time in a metric fashion, the planner makes use of a *temporal database*. As with most causal planners, PS begins with an incomplete plan and attempts to expand it into a complete plan by posting additional constraints in the database.

These constraints originate from the goals and from constraint templates stored in a domain model of the spacecraft. The temporal database and the facilities for defining and accessing model information during search are provided by the Heuristic Scheduling Testbed System (HSTS) [7]. 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 PS architecture. Additional details on the planner algorithm and its correctness can be found in [6]. The model describes the set of actions, how goals decompose into actions, the constraints among actions, and resource utilization by the actions. For instance, the model will encode constraints such as "do not take MICAS (camera) images while thrusting" or "ensure that the spacecraft does not slew when within a Deep Space Network communication window". These constraints are encoded in a stylized and declarative form called the Domain Description Language (DDL).

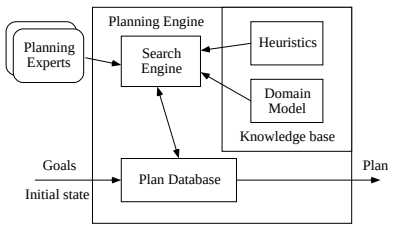

Figure 2. Planner/Scheduler Architecture

Each subsystem in the model is represented in the PS database as a set of dynamic *state variables* whose value is tracked over time. Each dynamic state variable can assume one or more values. A token is associated with a value of a state variable occurring over a finite

time interval. Each value has one or more associated compatibilities, i.e., patterns of constraints *between* tokens. A legal plan will contain a token of a given value only if all temporal constraints in its compatibilities are satisfied by other tokens in the plan. 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))))
```

Figure 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.

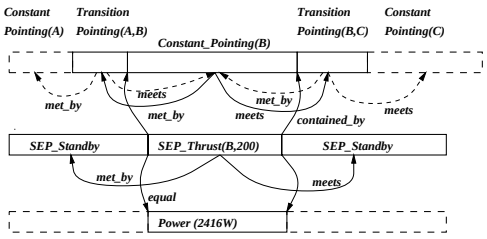

Figure 4. A Plan Fragment

The state-variable approach to modeling is also driven by strong software engineering principles. In a complex domain with different individuals and organizations with varying expertise, state-variables provide disparate and an object-oriented view of the same domain model across organizational boundaries.

Efforts are currently underway to use model-checking techniques to provide full coverage of the model's characteristics and also to automatically generate the heuristics from a given model of the domain. This will further allow mission designers and systems staff to build robust and complex models on their own without relying on the AI technologists themselves. Additional details about the planner can be found in [6], [7], [9] and [10].

2.2 Diagnosis and Repair

We refer to the diagnosis and repair engine of RA as MIR, for Mode Identification and Reconfiguration. MIR eavesdrops on commands that are sent to the on-board hardware managers by EXEC. As each command is executed, MIR receives observations from spacecraft

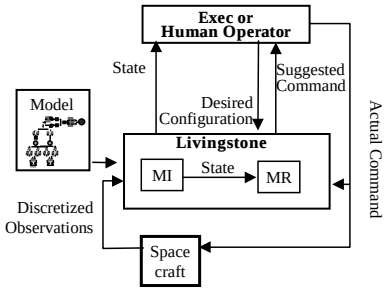

Figure 5. Livingstone Processing Cycle

sensors, abstracted by monitors in lower-level device managers such as for attitude control, bus controller, and so on. MIR uses an inference engine called Livingstone to combine these commands and observations with declarative models of the spacecraft's components to determine the current state of the system (Mode Identification) and report it to EXEC. EXEC may then request that Livingstone return a set of commands that will recover from a failure or move the system to a desired configuration (Mode Reconfiguration). Figure 5 illustrates the data flow between a spacecraft, EXEC and Livingstone.

MI is responsible for identifying the current operating or failure mode of each component in the spacecraft, allowing EXEC to reason about the state of the spacecraft in terms of component modes, rather than in terms of low-level sensor values. MR is responsible for suggesting reconfiguration actions that move the spacecraft to a configuration that achieves all current goals as required by PS and EXEC, supporting the run-time generation of novel reconfiguration actions. Although in RA, Livingstone is only used to recover following a component failure, its MR capability can be used to derive simple actions to reconfigure the spacecraft at any time. Thus Livingstone can be viewed as a discrete model-based controller in which MI provides the sensing component and MR provides the actuation component. Livingstone uses a single set of models and core algorithms to provide both the MI and MR functions.

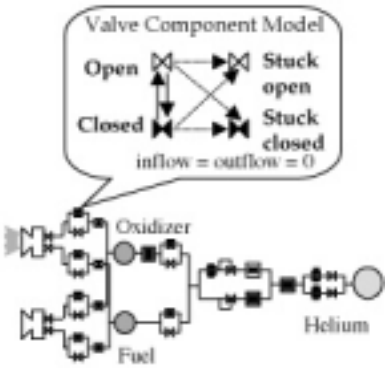

Figure 6. Livingstone Model of the Cassini Main Engine Subsystem

To use Livingstone, one specifies how the components of interest are connected. For each type of component, one then specifies a finite state machine that provides a description of the component's nominal and failure behavior. Figure 6 graphically depicts a Livingstone model of the Cassini spacecraft main engine subsystem. An important feature is that the behavior of each component state or mode is captured using abstract, or qualitative, models ([18] and [13]). These models describe qualities of the spacecraft's structure or behavior without the detail needed for precise numerical prediction, making abstract models much easier to acquire and verify than quantitative engineering models. Examples of qualities captured are

the power, data and hydraulic connectivity of spacecraft components and the directions in which each thruster provides torque. While such models cannot quantify how the spacecraft would perform with a failed thruster for example, they can be used to infer which thrusters are failed given only the signs of the errors in spacecraft orientation. Such inferences are robust since small changes in the underlying parameters do not affect the abstract behavior of the spacecraft.

Livingstone's abstract view of the spacecraft is supported by a set of fault protection monitors that classify spacecraft sensor output into discrete ranges (e.g. high, low nominal) or symptoms (e.g. positive X-axis attitude error). One objective of the RA architecture was to make basic monitoring capability inexpensive so that the scope of monitoring could be driven from a system engineering analysis instead of being constrained by software development concerns. To achieve this, monitors are specified as a dataflow schema of feature extraction and symptom detection operators for reliably detecting and discriminating between classes of sensor behavior. The software architecture for sensor monitoring is described using domain-specific software templates from which code is generated. Finally, all symptom detection algorithms are specified as restricted Harel state transition diagrams reusable throughout the spacecraft. The goals of this methodology are to reuse symptom classification algorithms, reduce the occurrence of errors through automation and streamline monitor design and test. Models are always incomplete in that they have an explicit unknown failure mode. Any component behavior that is inconsistent with all known nominal and failure modes is consistent with the unknown failure mode. In this way, Livingstone can infer that a component has failed, though the failure was not foreseen or was simply left unmodeled because no recovery is possible. By modeling only to the level of detail required to make relevant distinctions in diagnosis (distinctions that prescribe different recoveries or different operation of the system), we can describe a system with qualitative "common-sense" models that are compact and quite easily written.

Livingstone uses algorithms adapted from model-based diagnosis [17] to provide the above functions. Following de Kleer and Williams [1], MI uses a conflict directed best-first search to find the most likely combination of component modes consistent with the observations. Analogously, MR uses the same search to find the least-cost combination of commands that achieve the desired goals in the next state. Furthermore, both MI and MR use the same system model to perform their function. The combination of a single search algorithm with a single model, and the process of exercising these through multiple uses, contributes significantly to the robustness of the complete system.

The use of model-based diagnosis algorithms immediately provides Livingstone with a number of additional features. First, the search algorithms are sound and complete, providing a guarantee of coverage with respect to the models used. Second, the model building methodology is modular, which simplifies model construction and maintenance, and supports reuse. Third, the algorithms extend smoothly to handling multiple faults and recoveries that involve multiple commands. Fourth, while the algorithms do not require explicit fault models for each component, they can easily exploit available fault models to find likely failures and possible recoveries. Additional technical details about Livingstone can be found in [18].

2.3 The Smart Executive

The Smart Executive (EXEC) is a multi-threaded, reactive commanding system. The EXEC runs as a multi-threaded process that is capable of asynchronously executing commands in parallel and

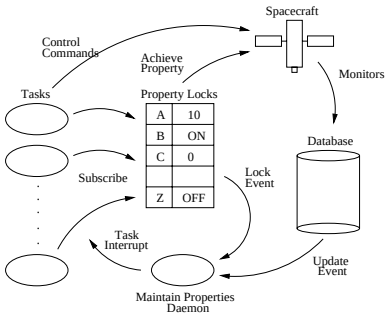

Figure 7. Smart Executive Architecture

is responsible for sending the appropriate commands to the various flight systems it is managing. It can replace the traditional spacecraft execution engine, or it can be used in conjunction with a traditional sequencer to command a complex subsystem such as an interferometer. In addition to these capabilities, the EXEC is capable of:

- Simultaneously achieving and maintaining multiple goals, i.e., system states, by monitoring the success of commands it issues and reactively re-achieving states that are lost.
- Conditional sequencing. Commands can be dependent on conditions that occur at execution time.
- Event-driven commanding, as opposed to traditional sequencers that are time-driven. For example, taking a sequence of pictures based on the results of monitoring a range sensor.
- High-level commanding and run-time task expansion. EXEC provides a rich procedural language, Execution Support Language (ESL) [4] in which spacecraft software/model developers define how complex activities are broken up into simpler ones. A procedure can specify multiple alternate methods for goal achievement to increase robustness.
- Sequence recovery. In the event that a command in an executing sequence fails, EXEC suspends execution of the failed sequence and attempts a recovery, either by executing a pre-specified recovery sequence such as reissuing the command or consulting a recovery expert, e.g., MIR. Once the desired state of the failed command is achieved, the suspended sequence is restarted.
- Temporally-flexible sequence (or plan) execution. In order to decrease the probability of a sequence failing, time ranges can be specified for executing and achieving the desired state for each command.
- Resource Management. EXEC manages abstract resources by monitoring resource availability and usage, allocating resources to tasks when available, making tasks wait until their resources are available, and suspending or aborting tasks if resources become unavailable due to failures. These tasks may compete for system resources within the constraints not already resolved by the Ground or the planner.

See [4] and [5] for a more detailed discussion. Figure 7 illustrates key functions of EXEC.

EXEC achieves multi-tasking through the use of property locks that could be used to maintain certain properties of the system. For example, if a task commands a switch ON, the switch property will be locked ON. Monitors (and MIR) determine if it is consistent to believe that the switch is ON. Since EXEC stores this state in its state database, should the inferred state of the switch change, the database will be updated and an event created, signaling a change. If the signaled event violates a property lock, an EXEC property thread interrupts those tasks that subscribed to that property lock. It will

then attempt to achieve the state of the switch being ON using its own recovery mechanism or by consulting a recovery expert, such as MR. If the switch cannot be turned ON in time, a hard deadline that is being tracked may be missed, so EXEC commands the spacecraft into a safe, wait state while it requests a new plan from the planner that takes into account that the switch cannot be turned ON.

Recoveries may be as simple as sending another command to turn a switch ON, or may be complex, such as when multiple subsystems are tightly coupled. For example, consider two coupled DS1 subsystems: the engine gimbal and the solar panel gimbal. A gimbal enables the engine nozzle to be rotated to point in various directions without changing the spacecraft orientation. A separate gimbal system enables the solar panels to be independently rotated to track the sun. In DS1, both sets of gimbals communicate with the main computer via a common gimbal drive electronics (GDE) board. If either system experiences a communications failure, one way to reset the system is to power-cycle the GDE. However, resetting the GDE to fix one system also resets the communication to the other system. In particular, resetting the engine gimbal, to fix an engine problem, causes temporary loss of control of the solar panels. Thus, fixing one problem can cause new problems. To avoid this, the recovery system needs to take into account global constraints from the nominal schedule execution, rather than just making local fixes in an incremental fashion, and the recovery itself may be a sophisticated plan involving operations on many subsystems.

EXEC and its commanding language, ESL, are currently implemented using multi-threaded Common LISP. A new version of EXEC is currently under development in C/C++. The internal EXEC code is designed in a modular, layered fashion so that individual modules can be designed and tested independently. Individual generic device knowledge for RAX is implemented based on EXEC's library of device management routines, to support addition of new devices and reuse of the software on future missions. More details about EXEC can be found in [4], [5] and [14].

3 Conclusion

The primary goal of Remote Agent on DS1 was to demonstrate that Artificial Intelligence based technologies could achieve high-level autonomous control of a spacecraft including:

- goal-oriented commanding;
- closed-loop planning and execution;
- spacecraft state inferencing and failure detection;
- closed-loop model-based failure diagnosis and recovery;
- on-board re-planning as a response to unrecoverable failures; and
- system-level fault protection.

Familiarizing the spacecraft engineering community with these technologies and laying the foundation for more extensive applications of RA were also important goals. These goals were achieved by the design of RA, its integration with the DS1 flight software on spacecraft testbeds, its layered testing, two operational readiness tests with ground control personnel, and successful commanding of the spacecraft in May 1999.

Future efforts using the RA involve extensions to the architecture towards domain-model unification and providing *continuous planning* capabilities [8]. The applications envisaged range from distributed spacecraft control, control of biological experiments for the International Space Station, instrument commanding for interferometry and on-board diagnosis for the next generation of reusable launch vehicles.

As a result of the Remote Agent Experiment, we believe that the willingness of NASA missions to deploy highly-autonomous systems has increased. Moreover, NASA has recognized the Remote Agent by bestowing the Software of the Year award, the agency's highest for technical achievement.

Acknowledgments

We gratefully acknowledge the contributions of all the members of the Remote Agent and DS1 teams. This paper describes work performed at the NASA Ames Research Center and at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration.

REFERENCES

[1] Johan de Kleer and Brian C. Williams, 'Diagnosing Multiple Faults', *Artificial Intelligence*, **32**(1), (1987).

[2] Douglas E. Bernard et.al, 'Design of the Remote Agent Experiment for Spacecraft Autonomy', in *Proceedings of the IEEE Aerospace Conference 1999, Snowmass, CO*, (1999).

[3] P. Pandurang Nayak et.al, '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*, (1999).

[4] Erann Gat, 'ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents', in *Procs. of the AAAI Fall Symposium on Plan Execution*, ed., Louise Pryor. AAAI Press, (1996).

[5] Erann Gat and Barney Pell, 'Abstract Resource Management in an Unconstrained Plan Execution System', in *Proceedings of the IEEE Aerospace Conference 1998, Snowmass, CO*, (1998).

[6] Ari Jonsson, Paul Morris, Nicola Muscettola, Kanna Rajan, and Ben Smith, 'Planning in Interplanetary Space: Theory and Practice', in *Proceedings of the 5th International AIPS, Breckenridge, CO*, (2000).

[7] Nicola Muscettola, 'HSTS: Integrated Planning and Scheduling', in *Intelligent Scheduling*, eds., M. Zweben and M. Fox, 169–212, Morgan Kaufman, (1994).

[8] Nicola Muscettola, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt, 'A Unified Approach to Model-Based Planning and Execution', in *Proceedings of 6th Int. Conf. on Intelligent Agent Systems, Venice, Italy*, (2000).

[9] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian William, 'Remote Agent: To Boldly Go Where No AI System Has Gone Before', *Artificial Intelligence*, **103**(1-2), 5–48, (August 1998).

[10] Nicola Muscettola, Ben Smith, Charles Fry, Steve Chien, Kanna Rajan, Gregg Rabideau, and David Yan, 'On-Board Planning for Autonomous Spacecraft', in *Proceedings of the Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS)*, Tokyo, Japan, (August 1997).

[11] Remote Agent Web page, 'http://rax.arc.nasa.gov'.

[12] 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', *Autonomous Robotics*, **5**(1), (1998).

[13] Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P.Pandurang Nayak, Christian Plaunt, and Brian Williams, 'A Hybrid Procedural/Deductive Executive for Autonomous Spacecraft', (1997).

[14] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith, 'Robust Periodic Planning and Execution for Autonomous Spacecraft', in *Proceedings of the IJCAI 97*.

[15] Kanna Rajan, Mark Shirley, William Taylor, and Bob Kanefsky, 'Ground Tools for Autonomy in the 21st Century', in *Proceedings of the IEEE Aerospace Conference, Big Sky,MT*, (March 2000).

[16] David Smith, Jeremy Frank, and Ari Jonsson, 'Bridging the Gap Between Planning and Scheduling', *Knowledge Engineering Review*, (15:1), (2000).

[17] Daniel S. Weld and Johan de Kleer, *Readings in Qualitative Reasoning About Physical Systems*, 1990.

[18] Brian C. Williams and P. Pandurang Nayak, 'A Model-Based Approach to Reactive Self-Configuring Systems', in *Procs. of AAAI-96*, pp. 971–978, Cambridge, Mass., (1996). AAAI Press.