The New Millennium Remote Agent: To Boldly Go Where No AI System Has Gone Before

Nicola Muscettola [‡]
Pandu Nayak [¶]
Barney Pell [¶]
Brian Williams [‖]
[§]

March 29, 1998

Abstract

The year 1997 marked the first chess program to defeat a world champion, the first time a mobile robotic explorer landed on another planet, and the fictitious birthday of HAL-9000 in the book "2001: A Space Odyssey". Autonomous Agents for space exploration has been a grand challenge for AI since its inception. This paper describes the New Millennium Remote Agent (NMRA) architecture for autonomous spacecraft control systems. The architecture supports challenging requirements of the autonomous spacecraft domain including highly reliable autonomous operations over extended time periods in the presence of tight resource constraints, hard deadlines, limited observability, and concurrent activity. A hybrid architecture, NMRA integrates traditional real-time monitoring and control with heterogeneous components for constraint-based planning and scheduling, robust multi-threaded execution, and model-based diagnosis and reconfiguration. Novel features of this integrated architecture include support for robust closed-loop generation and execution of concurrent temporal plans and a hybrid procedural/deductive executive.

NMRA will be demonstated as an onboard controller for Deep Space One (DS-1), the first flight of NASA's New Millennium Program (NMP), which will launch in 1998. As the first AI system to autonomously control an actual spacecraft, NMRA will enable the establishment of a "virtual presence" in space through an armada of intelligent space probes that autonomously explore the nooks and crannies of the solar system. Moreover, autonomous systems technology will help in manned exploration, possibly supporting humans and robots living off the land as part of a manned mission back to the moon or to Mars.

[§]Authors in alphabetical order.
[‡]Recom Technologies, NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.
[¶]RIACS, NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.
[‖]NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.

keywords: autonomous robots, agent architectures, action selection and planning, diagnosis, integration and coordination of multiple activities, fault protection, operations, real-time systems, modeling

1 Introduction

The melding of space exploration and robotic intelligence has had an amazing hold on the public's imagination, particularly in its vision of the future. For example, the science fiction classic "2001: A Space Odyssey" offered a future in which humankind was firmly established beyond Earth, within amply populated moon-bases and space-stations. At the same time, intelligence was firmly established beyond humankind through the impressive HAL9000 computer, created in Urbana, Illinois on January 12, 1997. Of course January 12th, 1997 has passed without a moon base or HAL9000 computer in sight. The International Space Station will begin its launch into space this year, reaching completion by 2002. However, this spacestation is far more modest in scope.

While this reality is far from our ambitious dreams for humans in space, space exploration is suprising us with a different future that is particularly exciting for robotic exploration, and for the information technology community that will play a central role in enabling this future:

> Our vision in NASA is to open the Space Frontier. When people think of space, they think of rocket plumes and the space shuttle. But the future of space is in information technology. We must establish a virtual presence, in space, on planets, in aircraft, and spacecraft.
>
> — Daniel S. Goldin, NASA Administrator, Sacramento, California, May 29, 1996

Achieving this ambitious goal requires a strong motive, mechanical means, and computational intelligence. We briefly consider the scientific questions that motivate space exploration and the mechanical means for exploring these questions, and then focus the remainder of this paper on our progress towards endowing these mechanical explorers with a form of computational intelligence that we call *remote agents*.

The development of a remote agent under tight time constraints has forced us to reexamine, and in a few places call to question, some of AI's conventional wisdom about the challenges of implementing embedded systems, tractable reasoning and representation. This topic is addressed in a variety of places throughout this paper.

1.1 Establishing a Virtual Presense in Space

Renewed motives for space exploration have recently been offered. A prime example is a series of scientific discoveries that suggest new possibilities for life in space. The best known example is evidence, found during the summer of 1996, suggesting that primitive life might have existed on Mars more than 3.6 billion years ago. More specifically, the recent discovery of extremely small bacteria on Earth, called nanobacteria, lead scientists to examine the Martian meteorite AlH84001 at fine resolution, where they found evidence suggestive

of "native microfossils, mineralogical features characteristic of life, and evidence of complex organic chemistry."[?] Extending a virtual presence to confirm or overturn these findings requires a new means of exploration that has higher performance and is more cost effective than traditional missions. Traditional planetary misions, such as the Galileo Jupiter mission or the Cassini Saturn mission, have price tags in excess of a billion dollars, and ground crews ranging from 100 to 300 personnel during the entire life of the mission. The Mars Environmental Survey mission concept (MESUR)[?] introduced a paradigm shift within NASA towards lightweight, highly focused missions, at a tenth of the cost, and operated by small ground teams. The viability of this concept was vividly demonstrated last summer through the success of NASA's Mars/MESUR Pathfinder mission. Likewise Sojourner, originating from micro-rover concepts developed by Dave Miller and others at JPL, represents a revolution in rover technology with a fraction of the weight and launch cost of multi-ton rovers like Ambler [?].

Pathfinder and Sojourner demonstrate an important mechanical means to achieving a virtual presence, but currently lack the onboard intelligence necessary to achieve the goals of more challenging missions. For example, operating Sojourner for its two month life span was extremely taxing for its small ground crew, with crew fatigue leading to commanding errors and loss of science opportunity. Future Mars rovers are expected to operate for over a year, emphasizing the need for the development of remote agents that are able to continuously and robustly interact with an uncertain environment.

Rovers are not the only means of exploring Mars. Another innovative concept is a Martian solar airplane, under study at NASA Lewis and NASA Ames. Given the thin CO_2 atmosphere on Mars, a plane flying a few feet above the Martian surface is like a terrestrial plane flying more than 90,000 feet above sea level. This height is beyond the reach of all but a few existing planes. Developing a Martian plane that can autonomously survey Mars over long durations, while surviving the idiosyncrasies of the Martian climate, requires the development of remote agents that are able to accurately model and quickly adapt to their environment.

A second example is the discovery of the first planet around another star, which raises the intriguing question of whether or not Earth-like planets exist elsewhere. To search for Earth-like planets, NASA is developing a series of interferometric telescopes, such as the New Millennium Deep Space Three (DS3) mission. These inferometers identify and categorize planets by measuring a wobble in a star, induced by its orbiting planets. They are so accurate that, if pointed from California to Washington DC, they could measure the thickness of a single piece of paper. DS3 achieves this requirement by placing three optical units on three separate spacecraft, flying in tight formation up to a kilometer apart. This extends the computational challenge to the development of multiple, tightly coordinated remote agents.

A final example is the question of whether or not some form of life might exist beneath Europa's frozen surface. In February of 1998, the Galilio mission identified features on Europa, such as a relatively smooth surface and chunky ice rafts, that lend support to the idea that Europa may have subsurface oceans, hidden under a thin icy layer. One of NASA's most intriguing concepts for exploring this subsurface ocean is an ice penetrator and a submarine, called a cryobot and hydrobot, that could autonomously navigate beneath

Europa's surface. This hydrobot would need to operate autonomously within an environment that is utterly unknown.

Taken together, these examples of small explorers, including micro-rovers, airplanes, formation flying interferometers, cryobots, and hydrobots, provide an extraordinary opportunity for developing remote agents that assist in establishing a virtual presence in space, on land, in the air and under the sea.

1.2 Requirements for Building Remote Agents

The level of onboard autonomy necessary to enable the above missions is unprecedented. Added to this challenge is the fact that NASA will need to achieve this capability at a fraction of the cost and design time of previous missions. In contrast to the billion dollar Cassini mission, NASA's target is for missions that cost under 100 million dollars, developed in 2-3 years, and operated by a small ground team. This ambitious goal is to be achieved at an Apollo-era pace, through the New Millennium Program's low cost, technology demonstration missions. The first New Millennium probe, Deep Space One (DS1), had a development time of only two and a half years and is scheduled for a mid-1998 launch.

The unique challenge of developing remote agents for controlling these space explorers is driven by four major properties of the spacecraft domain. First, a spacecraft must carry out *autonomous operations for long periods of time* with no human intervention. This requirement stems from a variety of sources including the cost and limitations of the deep space communication network, spacecraft occultation when it is on the "dark side" of a planet, and communication delays. For example, the Cassini spacecraft must perform its critical Saturn orbit insertion maneuver without any human assistance due to its occultation by Saturn.

Second, Autonomous operations must guarantee success, given *tight deadlines and resource constraints*. Tight deadlines that give no second chances stem primarily from orbital dynamics, and include examples such as executing an orbit insertion maneuver within a fixed time window or taking asteroid images during a narrow window around the time of closest approach. Tight spacecraft resources, whether renewable like power or non-renewable like propellant, must be carefully managed and budgeted throughout the mission.

Third, since spacecraft are expensive and are often designed for unique missions, spacecraft operations require *high reliability*. Even with the use of highly reliable hardware, the harsh environment of space can still cause unexpected hardware failures. Flight software must compensate for such failures by repairing or reconfiguring the hardware, or switching (to possibly degraded) operation modes. Providing such a capability is complicated by the need for *rapid failure responses* to meet hard deadlines and conserve precious resources, and due to *limited observability* of spacecraft state. The latter stems from limited on-board sensing, since additional sensors add weight, and hence increase mission cost. Furthermore, sensors are no more reliable, and often less so, than the associated hardware, thus making it difficult to deduce true spacecraft state.

Fourth, spacecraft operation involves *concurrent activity* among a set of *tightly coupled* subsystems. A typical spacecraft is a complex networked, multi-processor system, with one or more flight computers communicating over a bus with sophisticated sensors (e.g., star trackers, gyros, sun sensors), actuator subsystems (e.g., thrusters, reaction wheels, main

engines), and science instruments. These hybrid hardware/software subsystems operate as concurrent processes that must be coordinated to enable synergistic interactions and to control negative ones. For example, while a camera is taking a picture, the attitude controller must hold the spacecraft at a specified attitude, and the main engine must be off since otherwise it would produce too much vibration. Hence, all reasoning about the spacecraft must reflect this concurrent nature.

1.3 A Remote Agent architecture

Following the announcement of the New Millennium program in early 1995, spacecraft engineers from JPL challenged a group of AI researchers at NASA Ames and JPL to demonstrate, within the short span of five months, a fully capable remote agent architecture for spacecraft control. To evaluate the architecture the JPL engineers defined the New Millennium Advanced Autonomy Prototype (Newmaap), a simulation study based on NASA's Cassini Saturn mission, that retains its most challenging aspects. The Newmaap spacecraft is a scaled down version of Cassini, NASA's most complex spacecraft to date. The Newmaap scenario is based on the most complex mission phase of Cassini—successful insertion into Saturn's orbit even in the event of any single point of failure. The Remote Agent architecture developed for the Newmaap scenario integrated constraint-based planning and scheduling, robust multi-threaded execution, and model-based diagnosis and recovery. An overview of the architecture is provided in Section **??**. Additional details, including a description of the Newmaap scenario, may be found in [36]. The success of the Newmaap demonstration resulted in the Remote Agent being selected as a technology experiment on DS1. This experiment is currently scheduled for late 1998. Details of the experiment can be found in [2].

The development of the Remote Agent architecture also provided an important opportunity to reassess some of AI's conventional wisdom, which includes:

- "Generative planning does not scale up for practical problems."
- "[For reactive systems] proving theorems is out of the question" [1]
- "[Justification-based and Logical Truth Maintenance Systems] have proven to be woefully inadequate... they are inefficient in both time and space" [8]
- "[Qualitative] equations are far too general for practical use." [43]
- "Diagnostic reasoning from a tractable model is largely well understood. [However] we don't know how to model complex behavior..." [**?**].

We examine these statements in more detail later in the paper. But first we highlight the three important guiding principles underlying the design of the Remote Agent architecture.

1.4 Principles guiding the design of the Remote Agent

Many agent architectures have been developed within the AI community, particularly within the field of indoor and outdoor mobile robots. The Remote Agent architecture has three distinctive features. First, it is largely programmable through a set of compositional, declarative models. We refer to this as *model-based programming*. Second, it performs significant amounts of onboard *deduction and search* at time resolutions varying from hours to hundreds of milliseconds. Third, the Remote Agent is designed to provide *high-level closed-loop commanding*.

1.4.1 Model-based Programming

The most effective way to reduce software development cost is to make the software "plug and play," and to amortize the cost of the software across successive applications. This is difficult to achieve for the breadth of tasks that constitute an autonomous system architecture, since each task requires the programmer to reason through system-wide interactions to implement the appropriate function. For example, diagnosing a failed thruster requires reasoning about the interactions between the thrusters, the attitude controller, the star tracker, the bus controller, and the thruster valve electronics. Hence this software lacks modularity, and has a use that is very restricted to the particulars of the hardware. The one of a kind nature of NASA's explorers means that the cost of reasoning through system-wide interactions cannot be amortized, and must be paid over again for each new explorer. In addition, the complexity of these interactions can lead to cognitive overload by the programmers, causing suboptimal decisions and even outright errors.

Our solution to this problem is called *model-based programming*, introduced in [47]. Model-based programming is based on the observation that programmers and operators generate the breadth of desired functionality from commonsense hardware models in light of mission-level goals. In addition the same model is used to perform most of these tasks. Hence although the flight software itself is not highly reusable, the modeling knowledge used to generate this software *is highly reusable*.

To support plug and play, the Remote Agent is programmed, wherever possible, by specifying and plugging together declarative component models of hardware and software behaviors. The Remote Agent then has the responsibility of automating all reasoning about system wide interactions from these models. For example, the model-based diagnosis and recovery component of the Remote Agent uses a compositional, declarative, concurrent transition system model with a combination of probabilistic and deterministic transitions (see Section 5). Similarly, the planning and scheduling component is *constraint-based*, operating on a declarative domain model to generate a plan from first principles (see Section **??**). Even the executive component, which is primarily programmed using a sophisticated scripting language, uses declarative models of device properties and interconnections wherever possible; generic procedures written in the scripting language operate directly on these declarative models.

1.4.2 On-board deduction and search

Given the task of automating all reasoning about system interactions, a natural question is whether or not the Remote Agent should do this on-board in real-time or off-board at compile time. The need for fast reactions suggests that all responses should be precomputed. However, since our space explorers often operate in harsh environments over long periods of time, a large number of failures can frequently appear during mission critical phases. Hence pre-enumerating responses to all possible situations quickly becomes intractable. When writing flight software for traditional spacecraft, tractability is usually restored with the use of simplifying assumptions, such as using local suboptimal control laws, assuming single faults, ignoring sensor information, or ignoring subsystem interactions. Unfortunately, this can result in systems that are either brittle or grossly inefficient, which is one reason why so many human operators are needed within the control loop.

The difficulty of precomputing all responses and the requirement of highly survivable systems means that the Remote Agent must use its models to synthesize timely responses to anomalous and unexpected situations in real-time. This applies equally well to the high-level planning and scheduling component and to the low-level fault protection system, both of which must respond to time-critical and novel situations by performing deduction and search in real-time (though, of course, the time-scale for planning is significantly larger than for fault protection).

This goal goes directly counter to the conventional AI wisdom that robotic executives should avoid deduction within the reactive loop at all costs. This wisdom emerged in the late 80's after mathematical analysis showed that many, suprisingly simple, deductive tasks were NP Hard. For example, after proving that his formulation of STRIPS-style planning was NP Hard, David Chapman concluded [4]:

> "Hoping for the best amounts to arguing that, for the particular cases that come up in practice, extensions to current planning techniques will happen to be efficient. My intuition is that this is not the case."

On the flip side, what offers hope is the empirical work developed in the early 90's on hard satisfiability problems. This work found that most satisfiability problems can quickly be shown to be satisfiable or unsatisfiable [5; 44]. The suprisingly elusive hard problems lie at a phase transition from solvable to unsolvable problems. The elusiveness of hard problems, and the need to turn to random generation to find them, suggests that most real world problems are tractable. Furthermore, it suggests that if a deductive kernel is carefully designed and constrained, then it can perform significant deduction in real-time. For example, the diagnosis and recovery component of the Remote Agent adopts a RISC-like approach in which a wide range of deductive problems are reduced to queries on a highly tuned, propositional, best-first search kernel [48; 35]. The planning component exploits a set of assumptions about domain structuring to generate plans with acceptable efficiency using a simple search strategy and a simple language for writing heuristic control rules.

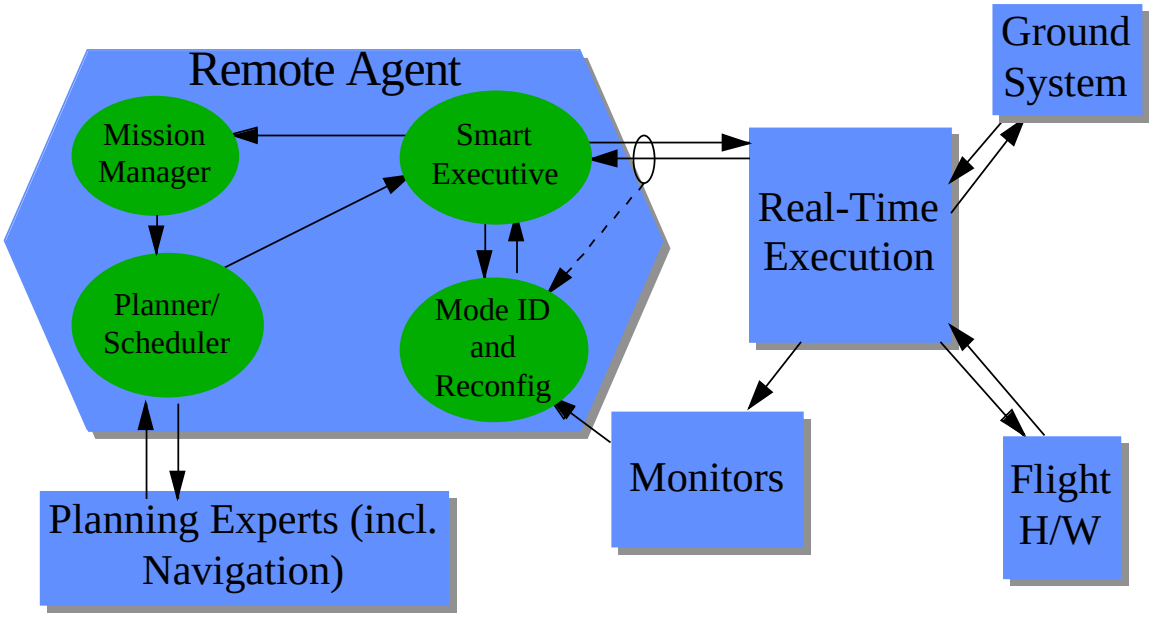

Figure 1: NMRA architecture embedded within flight software.

1.4.3 Goal-directed, closed loop commanding

It is striking to many layman that a mission like Cassini requires a ground crew of 100 to 300 personnel. Although Cassini's nominal mission is not simple, what drives the need for such a large team is the effort required to robustly respond to extraordinary situations. The need for extreme robustness is Remote Agent's most defining requirement.

The robustness of classical control systems dramatically improved through the concept of feedback control. Feedback control replaced direct commanding with a specification of intended behavior, in terms of a setpoint trajectory, and a feedback mechanism that commands the system until the error between actual and intend behavior is eliminated.

was dramatically improved

basic through the introduction of feedback control.

The Apollo 13 crisis offers a well known example that highlights the diverse efforts humans will take in order to achieve a mission.

an extreme example that highlights the kind of diverse

well known example that highlights the diversity

highlights the kind of

2 Remote Agent architecture

This section provides an overview of the Remote Agent (RA) architecture. The architecture was designed to address the domain requirements discussed in Section **??**. The need for autonomous operations with tight resource constraints and hard deadlines dictated the need for a temporal planner/scheduler (PS), with an associated mission manager (MM), that manages resources and develops plans that achieve goals in a timely manner. The need for

high reliability dictated the use of a reactive executive (EXEC) that provides robust plan execution and coordinates system fault protection, and a model-based mode identification and reconfiguration system (MIR) that enables rapid failure responses in spite of limited observability of spacecraft state. The need to handle concurrent activity impacted the representation formalisms used: PS models the domain with concurrently evolving state variables, EXEC uses multiple threads to manage concurrency, and MIR models the spacecraft as a concurrent transition system.

The RA architecture, and its relationship to the flight software within which it is embedded, is shown in Figure 1. When viewed as a black-box, RA sends out commands to the *real-time control* software (RT). RT provides the primitive skills of the autonomous system, which take the form of discrete and continuous real-time estimation and control tasks, e.g., attitude determination and attitude control. RT responds to high-level commands by changing the modes of control loops or states of devices. Information about the status of RT control loops and hardware sensors is passed back to RA either directly or through a set of *monitors*. The monitors discretize the continuous data into a set of qualitative intervals based on trends and thresholds, and pass the results back to the RA.

Planner/Scheduler (PS) and Mission Manager (MM): PS is an integrated temporal planner and resource scheduler [33] that uses constraint-based representations and heuristic guided chronological backtracking to develop plans. It is activated by MM when a new plan is desired by the EXEC. When requested by the EXEC, MM formulates short-term planning problems for PS based on a long-range mission profile. The mission profile is provided at launch and contains a list of all nominal goals to be achieved during the mission. MM determines the goals that need to be achieved in the next horizon (typically 2 weeks long) and combines them with the initial (or projected) spacecraft state provided by EXEC. This decomposition into long-range mission planning and short-term detailed planning enables the RA to undertake an extended diverse mission with minimal human intervention.

PS takes the plan request formulated by MM and produces a flexible, concurrent temporal plan. The plan constrains the activity of each spacecraft subsystem over the duration of the plan, but leaves flexibility for details to be resolved during execution. The plan contains activities and information required to monitor the progress of the plan as it is executed. The plan also contains an explicit activity to initiate the next round of planning.

Other onboard software systems, called planning experts, participate in the planning process by requesting new goals or answering questions for PS. For example, the navigation planning expert requests new goals in the form of pictures that are needed to update the understanding of spacecraft trajectory, and the attitude planning expert answers questions about estimated duration of specified turns and resulting resource consumption.

Smart Executive (EXEC): EXEC is a reactive plan execution system with responsibilities for coordinating execution-time activity. EXEC executes plans by decomposing high-level activities in the plan into commands to the real-time system, while respecting temporal constraints in the plan. EXEC uses a rich procedural language, ESL [**?**], to define alternate methods for decomposing activities.

EXEC achieves robustness in plan execution by exploiting the plan's flexibility, e.g., by

being able to choose execution time within specified windows or by being able to select different task decompositions for a high-level activity. EXEC also achieves robustness through closed-loop commanding, whereby it receives feedback on the results of commands either directly from the command recepient or by inferences drawn by the mode identification component of MIR. When some method to achieve a task fails, EXEC attempts to accomplish the task using an alternate method in that task's definition or by invoking the mode reconfiguration (MR) component of MIR.

When instructed to request a new plan by the currently executing plan, EXEC provides MM with the projected spacecraft state at the end of the current plan, and requests a new plan. If the EXEC is unable to execute or repair the current plan, it aborts the plan, cleans up all executing activities, and puts the controlled system into a stable safe state (called a standby mode). EXEC then provides MM the current state and requests a new plan while maintaining this standby mode until the plan is received.

Mode Identification and Reconfiguration (MIR): The MIR component of the RA is provided by Livingstone [48], a discrete model-based controller. Livingstone is distinguished by its use of a single declarative spacecraft model to provide all its functionality, and its use of deduction and search within the reactive control loop. Livingstone's sensing component, called mode identification (MI), tracks the most likely spacecraft states by identifying states whose models are consistent with the sensed monitor values and the commands sent to the real-time system. MI reports all inferred state changes to EXEC, and thus provides a level of abstraction to the EXEC, enabling it to reason purely in terms of spacecraft state, and not low-level sensor values.

Livingstone's commanding component, called mode reconfiguration (MR), uses the spacecraft model to find a least cost command sequence that establishes or restores desired functionality by reconfiguring hardware or repairing failed components. Within the RA architecture, MR is invoked by the EXEC with a recovery request that specifies a set of constraints to be established and maintained. In response, MR produces a recovery plan that, when executed by EXEC, moves the spacecraft from the current state (as inferred by MI) to a new state in which all the constraints are satisfied.

3 Planning and scheduling

The Planner/Scheduler (PS) of the Remote Agent provides the high-level, deliberative planning component of the architecture. It receives from MM and EXEC the initial spacecraft conditions and the goals for the next scheduling horizon. It produces a plan, i.e., a high-level program that EXEC must follows in order to achieve the required goals.

Figure 2 shows the structure of PS (the most detailed account to date of PS and its underlying architecture are [I-SAIRAS paper] and [HSTS paper]). A general-purpose *planning engine* provides a problem solving mechanism that can be reused in different application domains. The special-purpose *domain knowledge* completely characterizes the application. The planning engine consists of the *plan database* and the *search engine*. The plan database is provided by the Heuristic Scheduling Testbed System (HSTS) planning and scheduling

framework. The search engine calls the plan database to record the consequences of problem solving step and to require consistency maintenance and propagation services.

The search engine, or Iterative Refinement Scheduler (IRS), is a chronological backtracker that encodes a finite set of methods usable to extend a partial plan. For example, IRS can select a still unenforced temporal relation constraint. In this case IRS will add to the plan constraints to guarantee that the temporal relation is satisfied in any further plan refinement.

Programming the planning engine for a specific application requires both a description of the domain, the *domain model*, and methods for IRS to choose among branching alternatives during the search process, the *domain heuristics*.

The success of PS is largely dependent on the ability to provide a good model of the domain constraints. To do so, PS uses the Domain Description Language (DDL), part of the HSTS framework. DDL makes two strong assumptions on how to express a planning domain. Firstly, it structures the description of the system as a finite set of *state variables*. A plan describes the evolution of a system as as a set of parallel histories (timelines) over linear and continuous time, one per state variable. Secondly, it uses a unified representational primitive, the *token*, to describe both actions and state literals. As in [], a token extends over a metric time interval. The description of a system consists of constraints between tokens that must be satisfied in a plan for it to represent legal behaviors of the controlled system. We further discuss these structural assumptions in section 3.2.

PS can generate complex plans with performance acceptable for an on-board spacecraft application. We believe that this is due to the use of constraint posting and propagation as the primary problem solving method together with the restrictions on the topology of the constraint networks imposed by DDL's structural assumptions. PS can do so even when using a very simple search strategy (chronological backtracking) and a very simple heuristic language to program the search engine (rules that assign a numeric priority to plan flaws at the moment the flaws appear in the plan).

While the primary goal of PS certainly is to provide a reliable software flight software module for DS1, PS tries to remain as *true* as possible to concepts and techniques that have evolved from AI planning and scheduling research. As one would expect, this meeting of theory and practice showed us the need for significant extensions to classical planning and scheduling. However, we also believe that PS is a concrete example that some basic assumptions of AI planning are adequate and, in certain cases, necessary to tackle real-world, mission-critical applications. Most notably PS demonstrates that planning from a fine-grain, declarative model of the world is adequate for the solution of complex problems of practical significance. Furthermore, we believe that at the current time AI planning and scheduling techniques constitute possibly the only viable software engineering techniques for the complex problem of developing high-level commanding software for highly autonomous sytems. This bears great promises for the future of the technology.

We now discuss some of these points in more detail.

3.1 Non-classical aspects of the DS1 domain

A complex, mission-critical application like DS1 is a serious stress-test for classical AI planning and scheduling technology.

Figure 2: Structure of PS

The classical AI planning problem is the achievement of a set of goal conditions given an initial state and a description of the controlled system as a set of planning operators. Most classical AI planners use representations of the world derived from STRIPS, which sees the world as an alternation of indefinitely persistent states and instantaneous actions.

Classical schedulers see the world as a set of resources and a set of structured task networks, each task having a duration that is known a priori. In certain models, resources can be in different modes and require setup actions to transition between them; however setups are typically not included in classical scheduling formulations. Solving a problem involves allocating a start time and a resource to each task while guaranteeing that all deadlines and resource limits are satisfied.

The DS1 domain not only forces a view of the world that merges planning and scheduling, but also introduces the need for significant extensions to the classical perspective. Here is a quick review of the types of constraints on system dynamics and the types of goals that PS must handle.

3.1.1 System dynamics

To describe the dynamics of the spacecraft hardware and real-time software, we find the need to express *state/action constraints* (e.g., preconditions such as "To take a picture, the camera must be on"), *continuous time* and the management of *finite resources* (such as on-board electric power). Classical planning or classical scheduling cover all of these aspects. However, there are other modeling constraints that are equally important but outside the classical perspective.

- *persistent parallel threads*: separate system components evolve in a loosely coupled manner. This can be represented as parallel execution threads that may need coordination on their relative modes. Typical examples of such threads are various control loops (e.g., Attitude Control and Ion Propulsion System Control) that can never terminate but only switch between different operational modes.

- *functional dependencies*: several parameters of the model are best represented as functions of other parameters. For example, the duration of a spacecraft turn depends on the pointing direction from which the turn starts and the one where the turn ends. The exact duration of a turn is not known a priori but can only be computed after the turn has been inserted between source and destination in the final plan. Tracking the value of partially instantiated functional dependencies in the plan is necessary in order for the planner to evaluate progress and limit backtracking.

- *continuous quantities*: besides time, the planner must keep track of the status of other continuous quantities over time. These include renewable resources like battery charge or data volume and non-renewable resources like propellant levels. For example, in DS1 the Ion Propulsion System (IPS) engine accumulates thrust over long periods of time (on the order of months). During thrust accumulation, several other activities must be executed that require the engine to be shut down while the activity is going on. Between interruptions, however, the plan must keep track of the previously accumulated amount of thrust so as not to over-shoot or under-shoot the total requested thrust.

- *planning experts*: it is unrealistic to expect that all aspects of the domain will be encoded in PS. PS is only one part of a large, multi-team software development effort. Other teams have the expertise and, often, already available sophysticated software to effectively model subsystem behaviors and mission requirements. PS must be able to effectively communicate information with these third-party software modules, the planning experts. An example of planning expert in DS1 is the Navigation Expert which manages the spacecraft trajectory. The Navigation Expert is in charge of feeding PS with request for beacon asteroid observations to determine the trajectory error; it also gives PS thrusting maneouver goals in order to maintain the desired trajectory.

3.1.2 Goals

The DS1 problem can only be expressed by making use of a disparate set of classical and non-classical goal types. Problem requirements include conditions on *final states* (e.g., "at the end of the scheduling horizon the camera must be off"), which are classical planning goals, and requests for *scheduled tasks* within given temporal constraints (e.g., "communicate with Earth only when one of the antennas of the Deep Space Network is visible from the spacecraft"), which are classical scheduling goals. Non-classical categories of goals include:

- *periodic goals*: for example, optical navigation activities are naturally expressed as a periodic function ("take asteroid pictures for navigation for 2 hours every 2 days plus/minus 6 hours"). PS must unfold this goal and schedule picture taking activity while taking into account interactions with other planned tasks or requested resource modes.

- *information seeking goals*: this arises in the interaction with planning experts and are not unlike those that can be found in the softbot domain (e.g., "Ask the on-board navigation planning expert for the most up-to-date thrusting profile").

- *quantity accumulation*: these arise in the handling of continuous resources. For example, in DS1 a goal expresses the requested thrust accumulation as a duty cycle, i.e., the percentage of the scheduling horizon during which the IPS engine is thrusting. PS will choose the specific time intervals during which IPS will be actually thrusting. It will do so by trading off IPS requireements with those of other goals.

- *default goals*: they specify conditions that the system must satisfy when not trying to achieve any other goal. For example, in order to facilitate possible emergency communications the spacecraft should keep the High Gain Antenna pointed to Earth whenever there is no other goal requiring it to point in a different direction.

3.2 Domain Structure Assumptions in PS

We mentioned that PS makes two strong structural assumptions on how to represent domain models. We call them the *state variable assumption* and the *token assumption*. We now discuss both of these in more detail.

1. *state variable assumption:* the evolution of any system over time is entirely described by the values of a finite set of state variables.

State variables are a generalization of resources as used in classical scheduling. In scheduling an evolution of the system is a description of task allocation to resources. Similarly, in PS any literal used inside a plan must be associated to a state variable. The literal represents the value assumed by the state variable at a given time, and a state variable can assume one and only one value at any point in time. Building a plan involves determining a complete evolution of all system state variables over a scheduling horizon of finite duration.

At first, structuring a model with a finite set of state variables could appear quite restrictive. By representing plans as partially ordered graphs of actions and states, classical planning seems more permissive. Also classical planning does not impose limits on the number of literals that can appear in the descriptor of a state. However, a more in-depth look shows that this need not be the case. In the first place, it is quite easy to identify state variables in domains typically addressed in classical planning. For example, in the "monkey and bananas" world all actions and state literals can be assigned as the values of one or more of the following state variables: the location of the monkey, the location of the block, the location of the bananas and the elevation of the monkey (whether the monkey is on the floor, climbing on the block or on top of the block). In the second place, recent results in planning research seem to suggest that planners that use representational devices similar to state variables can seriously outperform planners that do not.

1. *token assumption:* no distinction needs to be made between representational primitives for actions and states. A single representational primitive, the *token*, is sufficient to describe the evolution of a system over time.

This structural assumption challenges a fundamental tenet of classical planning: the dichotomy between actions and states. To illustrate why we believe this dichotomy is problematic, we consider an example drawn from the spacecraft operations domain.

The attitude of a spacecraft, i.e., its orientation in three-dimensional space, is supervised by a closed-loop Attitude Control System (ACS). When asked to achieve or maintain a certain attitude, ACS determines the discrepancy between the current and the desired attitude. It then appropriately commands the firing of the spacecraft thrusters if the discrepancy is higher than a maximum acceptable error. This cycle is continuously repeated until the attitude error is acceptable. When controlled by ACS, the spacecraft can be in one of two possible modes:

1. *Turning (x, y)*, i.e., changing attitude from an initial pointing x to a final pointing y;
2. *Constant_Pointing (z)*, i.e., maintaining attitude around a fixed orientation z.

If using a classical planning representation to model attitude, we would need to map these two modes into two different kinds of literals: *state* literals, representing persistent conditions, or *action* literals, representing change. The problem is that in spite of appearances it is by no means easy to choose the mapping between system modes and states/actions. Most people

would probably find it natural to map *Constant_Pointing (z)* to a state literal and *Turning (x, y)* to an action literal. This is certainly reasonable if one focuses on the value over time of the actual orientation of the spacecraft.

However, we may want to take a different perspective and consider the level of "activity" of the thrusters during attitude control. Indeed, it is the thrusters that are commanded by ACS and the attitude of the spacecraft is a consequences of thruster actions. In this case, the situation is different. It so happens that thrusters are usually more active the lower the acceptable error in attitude is. In fact, thrusters are fired more frequently while maintaining a *Constant_Pointing (z)* state with a very low error tolerance than while executing a *Turning (x, y)*, where it may be sufficient to fire the thrusters at the beginning of the turn to start it and at the end of the turn to stop it. In this case, one would conclude that in fact both *Turning (x, y)* and *Constant_Pointing (z)* would be best represeted as actions.

The opposite perspective is also possible. If we focus on what EXEC does when executing literals present in the plan, we can see that EXEC does nothing more than communicating to ACS the appropriate control law and set point that will cause the required spacecraft attitude behavior. From this point of view, it would be reasonable to see both *Constant_Pointing(z)* and *Turning (x, y)* are two different parameter setting for the ACS control system, conceptually best represented with state literals.

In this example the distinction between actions and states is not clear. As a consequence, PS takes a radical view and gives all literals the same status. More precisely, a plan literal always describes some process (either dynamic or stationary) that occurs over a period of time of non-negative duration. To purposefully remove any reference to the state/action dicothomy we use the neutral term *token* to refer to such temporally scoped assertions.

The main consequence of this assumption is that all representational primitives are uniformly available for all tokens of all types. For example, Figure 3 describes the conditions that have to be satisfied in the plan in order for the DS1 Microelectronics Integrated Camera And Spectrometer (MICAS) to take an image. It is similar to temporally scoped operators used in temporal planning approaches []. In our framework, however, similar constraints can also be imposed on "state" tokens like `MICAS.action_sv = Idle`. We call these constraint patterns *compatibilities* to emphasize their different nature from planning operators. Additionally, PS models can express functional duration constraints both on "actions" (e.g., the duration of *Turning (x, y)* depends on the angle between x and y) and on "states" (e.g., the maximum duration of *Constant_Pointing (z)* depends on the relative orientation of the Sun with respect to z since this determines the satisfaction of thermal constraints during on sun exposure for sensitive parts of the spacecraft).

3.3 Plans as Constraint Networks

PS plans are effectively programs that EXEC interprets at run time to generate a single, acceptable and consistent behavior for the spacecraft. However, to ensure execution robustness plans should as much as possible avoid being single, completely specified behaviors. They should instead compactly describe a *behavior envelope*, i.e., a set of possible behaviors. EXEC can incrementally select the most appropriate behavior in the envelope while responding to information that becomes available only at execution time.

```
MICAS.actions_sv =
  Take_Image (?id, ?orientation, ?priority, ?exp_time, ?settings)
  {
    :parameter_functions
      ?_duration_ <- Compute_Image_Duration (?exp_time, ?settings);

    :temporal_relations
      met_by
        MICAS.action_sv = Idle;
      meets
        MICAS.action_sv = Idle;
      equal
        Power.availability_sv =
              DELTA { Used <- Used + 140.0;
                      Available <- Available - 140.0;};
      contained_by
        Spacecraft.attitude_sv =
    Constant_Pointing (?orientation);
      contained_by
        MICAS.health_sv = MICAS_Available;
      contained_by
        MICAS.mode_sv = Ready;
  }
```

Figure 3: Taking a picture with the on-board MICAS camera.

PS satisfies this requirement by representing plans as constraint networks. For example, start and end times of tokens are integer-valued variables interconnected into a simple temporal constraint network []. Codesignations relate parameters that must assume the same value for any plan execution. Other functional dependencies can also be represented. For example, tokens that describe thrust accumulation with the IPS engine contain constraints that relate the initial accumulation (due to previous thrust accumulation tokens), the final accumulation and the duration of the token. During plan construction, when PS tries to enforce compatibility constraints, it actually posts portions of a constraint network in the plan database. The plan database can then be required to enforce consistency checking by propagate the new constraints to the rest of the network. when the constraint network is consistent, constraint propagation deduces acceptable ranges of values for each variable.

Plans are intrinsically *flexible*. During plan execution, EXEC interprets the plan's constraint network in order to select specific values for the plan variables. For example, if the plan specifies an acceptable range for the start time of a token, EXEC will have the freedom to start token execution at any one of the range values. This decision will affect the value range for the start or end of other, as yet unexecuted tokens. To adjust value ranges, EXEC must be able to execute constraint propagation at run time.

EXEC's constraint propagation has very different requirements than that of PS. The time EXEC needs to propagate constraints directly affects EXEC's *responsiveness*, i.e., the ability to give real-time guarantees on the exact time at which EXEC will actually start or terminate the execution of a token. This is because EXEC cannot discriminate between events whose temporal distance is smaller than the time needed to process a single event (i.e., propagate constraint, terminate all tokens ending with the event and start all tokens starting with the event). This lower bound on event discrimination is a hard limit on reactiveness [34]. One can show that different equivalent constraint networks (i.e., constraint networks that can yield the same set of variable/value assignments) can yield very different constraint propagation requirements at execution time. PS solves the reactiveness problem for temporal constraint networks by post-processing the network and transforming it into an equivalent, *minimal* one. Minimality here means that EXEC needs only perform the minimum possible amount of constraint propagation while still ensuring complete accessibility to all consistent behaviors that can be generated by the constraint network.[][].

3.4 On the economical feasibility of generative planning

Figure 4 outlines PS search process. If the partial plan in the plan database has "flaws", PS non-deterministically selects one and extends the plan constraint network to fix it. Then the plan database performs an arc-consistency propagation to detect inconsistencies and restrict variable value ranges. If propagation detects an inconsistency, then PS cronologically backtracks. When the plan database contains no more flaws, a plan is returned.

PS recognizes several kind of flaws. One example of *unscheduled goal token* flaw arises when the Navigation Expert returns requests for beacon asteroid images. These are represented as instances of tokens for which PS has not yet found a legal position on a state variable. If PS cannot find a legal position for a token, then the observation goal is rejected. The *underconstrained variable value* flaw arises in two cases: (1) the value range for

Figure 4: PS problem solving cycle

a variable does not permit PS to uniquely select a compatibility descriptor for a token; (2) arc consistency is insufficient to guarantee that EXEC will be able to generate a consistent behavior at execution time.

PS is a purely generative planner. Essentially, the problem solving cycle alternates between posting compatibility constraints and restricting the value range of some variable. Posting compatibility constraints is analogous to *subgoaling* in classical generative planners, while restricting ranges is analogous to the *value selection* in constraint propagation search.

With these simple devices PS can generate plans of up to 156 tokens and 186 temporal constraints between tokens. The search tree required to find a solution has 649 nodes and a search efficiency of about 64% where efficiency is measured by the ratio between the depth of the search tree and the number of expanded nodes (a search efficiency of 100% indicates no backtracking).

PS does not use pre-compiled token networks but assembles the overall plan from atomic components. This fact differenciate PS from all the practical applications of planning technology to date [] [] []. These systems rely on Hierarchical Task Network (HTN) planning, in which most of the power comes from hand-generated task networks that are then patched together into an overall plan. The strong negative correlation between generative planning and successful applications has led to the hypothesis that "precondition achievement planning has limited utility with respect to the automatic solution of economically viable planning problems". We will now discuss to which extent our work questions the validity of this hypothesis.

Although precompiling token networks into HTN could be a powerful problem solving technique, our choice of pure generative planning is not accidental. Firstly, the characteristics of the DS1 domain are not necessarily amenable to HTN planning because in DS1 the planner must address a wide range of goal types, task decomposition hierarchies tend to be shallow, and the various components of the controlled system can operate in several possible modes. The result is that the number of required task decompositions is potentially so large that it may be impractical to generate them and store them. Secondly, and most importantly, the HTN formalism does not provide a strong separation between the encoding of the domain model and that of the problem solving heuristics. While the former is valid irrespective of the goals of a specific planning problem, the function of the latter is to ensure acceptable performance and quality for the solution of specific planning problems. We choose instead to clearly separate between domain model and heuristics. By using a less powerful problem solving control mechanism, the shortcomings of our basic modeling approach and our problem solving methods became more apparent and allowed us to more clearly focus on needed improvements. As we shall see in section 6, the separation between domain models and problem solving heuristics is crucial to facilitate validation and, ultimately, has a big impact on the acceptability of AI technologies for critical applications.

4 Executive

The main challenge for spacecraft autonomy is to operate the system reliably and over extended periods of time without intervention. To this end EXEC plays the main coordination

role as the intermediary between the other flight software modules, both internal and external to RA. Here we concerntrate on two main aspects of EXEC's behavior:

- **Periodic Planning over Extended Missions:** EXEC must periodically ask PS for new tasks and must coordinate PS operation with the other tasks in execution. Also, operations are not interrupted if capabilities are lost. EXEC will ask for a new plan by communicating to PS the available capabilities.
- **Robust Plan Execution:** EXEC must successfully execute plans in the presence of uncertainty and failures. The flexibility allowed by the plan is exploited by using a *hybrid procedural/deductive* execution strategy that performs *context-dependent* method selection guided by state inference based on *model-based diagnosis*. Local recovery from faults involves planning guided by constraints from the current plan execution context.

4.1 Periodic Planning over Extended Missions

4.1.1 Planning to plan

In the spacecraft domain, planning itself has informational preconditions (since planning relies on input from other agents, who often need to complete some activity before they have suitable input), state preconditions (it is hard to plan when too many things are changing quickly or unpredictably), and consumes scarce computational resources. Therefore, in RA invoking the planner is analogous to commanding other subsystems like propulsion or attitude control. Future planning activities appear in plans on a timeline and domain constraints inforced in the plan ensure that their resources and preconditions will all be achieved before planning is invoked. This aspect of *planning to plan* [41] can be considered a form of meta-planning [**?**], and addresses one of the significant outstanding goals in the Phoenix agent architecture [6].

RA's approach to planning to plan is illustrated in Figure 5. In this example, the PS model represents the constraint that the next round of planning should occur only after the navigation system has performed a new orbit calculation. This calculation relies on analysis of several pictures, so PS inserts into the plan the supporting imaging activities and the turns required to point the camera at the corresponding targets. During execution, EXEC will initiate the next round of planning when it executes the `planning` activity installed in the plan. Because of the constraints explicit in the plan, this will happen only after the activities required for planning have been successfully completed.

4.1.2 Concurrent Planning and Execution

Even at pre-scheduled times, the limited computational resources available for planning, combined with the difficulty of planning with severe resource limitations, cause each round of planning to take a long time to complete. Throughout this process, the spacecraft will still need to be operating with full capabilities. For example, with the current on-board processor capabilities it is reasonable to expect PS to take up to 8 hours to generate a plan for one week of operation. This sums up to about six percent of the total mission time spent in generating plans. However, to reach the designated targets IPS propulsion may need to

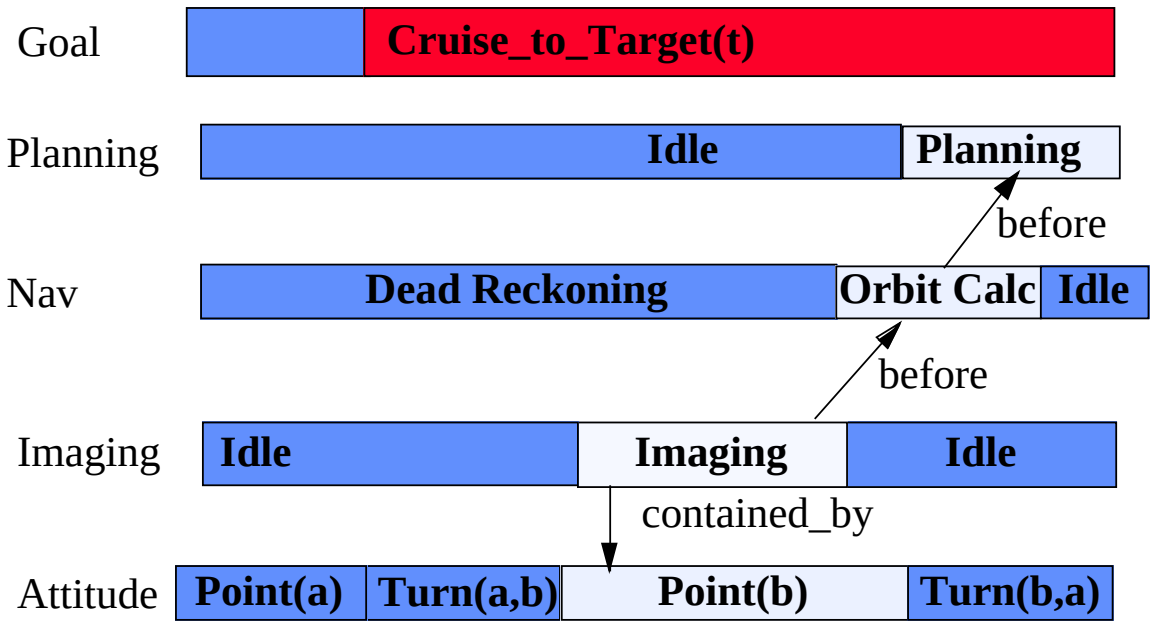

Figure 5: Sample Plan Fragment.

be operating with very duty cycles in eccess of 92% of the available time. Considering that a succesful mission requires the execution of other activites (such as scientific experiments and observations) and that often these require the IPS engine to be off, it may not possible to achieve the mission goals and avoid thrusting IPS at full force while PS is operating. Hence, EXEC operates concurrently to PS [41]. This involves tracking changes to the planning assumptions while planning, and using the currently-executing plan for prediction about activities which will happen while planning for a new period is underway.

4.1.3 Replanning with degraded capabilities

When operating over extended periods of time a spacecraft must face problems arising from aging: the capabilities of its hardware and control system may diminish over time. Once these failures are recognized through a combination of monitoring and diagnosis, EXEC will keep track of such degradation when commanding future planning cycles. For example, one fault in DS1 is for one of its thrusters to be stuck shut. The attitude control software has redundant control modes to enable it to maintain control following the loss of any single thruster, but an effect of this is that turns take longer to complete. When EXEC is notified of this permanent change by MIR, it passes health information back to the planner. The health information can also be used to update the models in PS. Capabilities can also be opportunistic re-established, such as may happen following a reboot or human-assisted repair procedures.

4.2 Robust Plan Execution

We have seen that in nominal operations EXEC invokes the planning machinery as a by-product of plan execution, and ensures that the agent is functioning on recent information. However, if execution fails before the planning activity is properly prepared and executed, the agent still needs a way to generate a plan and continue making progress on mission goals. RA addresses this problem as follows: if the EXEC is unable to execute or repair the current plan, it aborts the plan, cleans up all executing activities, and puts the controlled system into a stable safe state, called a *standby mode*, which serves (by design) as a well-defined invocation point for planning.

Hence, Figure 6 shows both major branches of the periodic planning and replanning cycle in RA. If plan execution proceeds to the point of a pre-planned planning activity, EXEC invokes the planner, continues executing while waiting for the new plan and smoothly installs the new plan into the current execution context. In the event of plan failure, the executive aborts all current activity, enters standby mode, requests a new plan from this well-defined state (possibly updating the planner about degraded capabilities) and starts executing the plan as soon as it receives it back from PS.

Note that establishing standby modes following plan failure is a costly activity with respect to mission goals, as it causes us to interrupt the ongoing planned activities and lose important opportunities. For example, a plan failure causing us to enter standby mode during the comet encounter would cause loss of all the encounter science, as there is no time to re-plan before the comet is out of sight. Such concerns motivate a strong desire for plan robustness, so that plan execution can continue successfully even in the presence of uncertainty and failures.

RA achieves robust plan execution through the following organization:

- Choose a high level of abstraction for planned activities so as to delegate as many detailed activity decisisions as possible to the procedural executive.
- Handle execution failures using a combination of robust procedures and deductive repair planning.

4.2.1 Delegating activity details to execution

Generation of plans with temporal flexibility follows from the architecture of PS as a constraint-based, least-commitment planner. A complementary source of plan robustness relies on careful knowledge representation for each domain. The approach is to choose an appropriate level of abstraction for activities planned by PS so as to leave as many details as possible to be resolved by EXEC during execution. A PS token is abstracted in the sense that it provides an *envelope* of resources (e.g., execution time allowances, maximum allocated power consumption) and synchronization constraints across envelopes. For each token EXEC has a task decomposition into more detailed activities that in the absence of exogenous failures are guaranteed by design to be executable within the resource envelopes.

An example from the DS-1 Remote Agent Experiment [2] (see Figure 7) illustrates this approach. A `delta-v` goal token requires the achievement a certain change (delta) in the velocity of the spacecraft. Velocity changes are achieved by thrusting the engine for some

Figure 6: Periodic Planning and Replanning Cycle.

Figure 7: Plan fragment for achieving a change in spacecraft velocity.

amount of time while pointing the spacecraft at a certain direction. A total velocity change is achieved via a series of shorter thrust segments, where between each segment the engine thrust is stopped while the spacecraft must be turned to the direction required by the next segment. There is a constraint that ACS be in thrust-vector-control (TVC) mode shortly after IPS has started thrusting and it must be in Reaction Control System (RCS) control mode upon termination of a thrusting activity.

Initiating a thrust activity involves a number of complex operations on the engine and there is considerable uncertainty about how long this initiation takes before thrust starts accruing. This translates into uncertainty about when to switch atittude control modes, how much thrust will be actually accrued in a given segment, and how many thrust segments are necessary to achieve the total desired thrust. RA follows the following approach to this problem. PS inserts thrust tokens into the plan which may not need to be executed. EXEC tracks how much thrust has been achieved, and only executes thrust tokens (and associated turns) for so long as thrust is actually necessary. Similarly, PS delegates to EXEC the coordination of activity details across subsystems that are below the level of visibility of the planner. In this example, we represent in EXEC's domain knowledge the constraint between the engine thrust activity and the control mode of the ACS. The result is that plan execution is robust to variations in engine setup time and in thrust achievement. It should be noted that this delegation of labor from PS to EXEC relies on many of the capabilities of a sophisticated procedural execution system [38; 23; 22].

4.2.2 Hybrid procedural/deductive executive

Another major cause of execution failure in the spacecraft domain is activity failure, often due to problems with the hardware.

The design of a spacecraft fault protection is complicated by the presence of coupling and interaction between tasks. In particular, the local recovery of a failed activity may require the use of an action (e.g., reset the control electronics of a device) that will negatively affec concurrent activities that are operating nominally (e.g., other devices that rely on continous operations of the same control electronics).

To avoid this, the recovery system needs to take into account global constraints from nominal schedule execution, rather than just making local fixes in an incremental fashion. This sort of interaction may require more complex and elaborate procedures that should be incorporated in a full recovery plan.

Examples like these drove the design of NMRA's hybrid execution system [39], which integrates a procedural executive based on generic procedures with a deductive model-based executive. A procedural executive (like RAPS [19], PRS [24], RPL [31], Interrap [32] and Golog [28]) provides sophisticated control constructs such as loops, parallel activity, locks, and synchronization which are used for robust schedule execution, hierarchical task decomposition, context-dependent method selection, and routine configuration management. A deductive executive provides algorithms for sophisticated state inference and optimal failure recovery planning. NMRA's integrated executive enables designers to code knowledge via a combination of procedures and declarative models, yielding a rich modeling capability suitable to the challenges of real spacecraft control. The interface between the two executives

Figure 8: Interacting subsystems in DS-1.

Figure 9: Simplified schematic of Cassini spacecraft propulsion system.

Figure 10: Livingstone architecture diagram

Figure 11: Different configurations that achieve thrust

[39] ensures both that recovery sequences are smoothly merged into high-level schedule execution and that a high degree of reactivity is retained to effectively handle additional failures during recovery.

5 Model-based mode identification and reconfiguration

The mode identification and reconfiguration component of the Remote Agent architecture is based on the Livingstone system [48]. Livingstone is a discrete model-based controller that sits at the nexus between the high-level feedforward reasoning of classical planning and scheduling systems, and the low-level feedback control of continuous adaptive methods (see Figure 10). It is a discrete controller in the sense that it constantly attempts to put the spacecraft hardware and software into a configuration that achieves a set point, called a *configuration goal*, using a sensing component, called *mode identification*, and a commanding component, called *mode reconfiguration*. It is model-based in the sense that it uses a single declarative, compositional spacecraft model for both MI and MR.

A configuration goal is a specification of a set of hardware and software configurations (or modes). More than one configuration can satisfy a configuration goal, corresponding to line and functional redundancy. The executive generates configuration goals in the process of plan execution and task decomposition. For example, the planner may require that the spacecraft accumulate a certain quantity of thrust over a period of time. The executive might decompose this activity into the configuration goal "fire main engine," which corresponds to spacecraft configurations in which an appropriate set of valves are open to enable propellant flow into the main engine and the low-level attitude control software is properly configured (see Figure 11).

Livingstone's sensing component, mode identification (MI), provides the capability to track changes in the spacecraft's configurations due to executive commands and component failures. MI uses the spacecraft model and executive commands to predict the next nominal configuration. It then compares the sensor values predicted by this configuration against the actual values being monitored on the spacecraft. Discrepancies between predicted and monitored values signal a failure. MI isolates and diagnoses the failure, thus identifying the actual spacecraft configuration, using algorithms adapted from model-based diagnosis [13; 14].

MI provides a variety of functions within the overall architecture. These include:

- Mode confirmation: Provide confirmation to the executive that a particular spacecraft command has completed successfully.
- Anomaly detection: Identify observed spacecraft behavior that is inconsistent with its expected behavior.

- Fault isolation and diagnosis: Identify components whose failures explain detected anomalies. In cases where models of component failure exist, identify the particular failure modes of components that explain anomalies.

- Token tracking: Monitor the state of properties of interest to the executive, allowing it to monitor plan execution.

When the current spacecraft configuration ceases to satisfy the currently active configuration goals, Livingstone uses its mode reconfiguration (MR) capability to identify a a set of control procedures that, when invoked, take the spacecraft into a new configuration that satisfies the goals.

The *mode reconfiguration* (MR) component of Livingstone is responsible for identifying a set of control procedures that when invoked take the spacecraft from the current state, to a lowest cost state that achieves a set of goal behaviors. MR can be used to support a variety of functions within the architecture, including:

- Mode configuration: Places the spacecraft in a least cost hardware configuration that exhibits a desired behavior.

- Recovery: Moves the spacecraft from a failure state to one that restores a desired function.

- Standby and Safing. In the absence of full recovery, places the spacecraft in a safe state while awaiting additional guidance from the high-level planner or ground operations team.

- Fault avoidance: Given knowledge of current, irreparable failures, finds alternative ways of achieving desired goals.

Three technical features of Livingstone are particularly worth highlighting. First, the long held vision of model-based reasoning has been to use a single central model to support a diversity of engineering tasks. As noted above, Livingstone automates a variety of tasks using a single model and a single core algorithm, thus making significant progress towards achieving the model-based vision. Second, Livingstone's representation formalism achieves broad coverage of hybrid discrete/continuous, software/hardware systems by coupling the concurrent transition system models underlying concurrent reactive languages [29] with the qualitative representations developed in model-based reasoning [46; 15]. Third, the approach unifies the dichotomy within AI between deduction and reactivity [1; 3], by using a conflict-directed search algorithm coupled with fast propositional reasoning. We now discuss these latter two points in more detail.

5.1 Representation formalism

Traditionally, model-based diagnosis has been formalized using first-order logic [42; 11]. First-order logic, with its expressivity and its clear declarative semantics, is certainly an appropriate formalism for precisely characterizing model-based diagnosis. However, it is wholly inappropriate as a representation formalism for building practical systems. On the

Figure 12: Transition system examples

one hand, its expressivity leads to computational intractability (first-order satisfiability is semi-decidable), precluding its use in a real-time system. On the other hand, first-order logic by itself, with its flat structure of constants, functions, and relations, provides little or no constraint in modeling.

The main approach to resolving the above limitations of first-order logic has been to use *constraint-based* modeling, e.g., see [7; 13; 45; 26], which directly addresses the issue of computational tractability. In addition, system models are built *compositionally* from individual component models and a specification of the connections between components. Each component model consists of a set of *modes*, corresponding to the different nominal and failure modes of the component. A set of constraints characterize the behavior of the component in each of its modes. The compositional, component-based nature of the modeling formalism enables plug-and-play model development, supports the development of complex large-scale models, increases maintainability, and enables model reuse.

5.1.1 Concurrent transition systems

While compositional constraint-based modeling is well suited for many model-based diagnosis applications, it has one major limitation: it has no model of *dynamics*, i.e., no model of transitions between modes. Modeling mode transitions is essential for Livingstone since it needs to track changes in spacecraft configurations and determine reconfiguration sequences.

We overcame the above limitation by coupling compositional constraint-based modeling with the *concurrent transition system* models used to model reactive systems [29]. In this formalism, each component is modeled as a transition system consisting of a set of modes with explicit transitions between modes. For example, Figure 12 shows the modes and transitions of a valve and a valve driver. As before, each mode is associated with a set of constraints that describe the component's behavior in that mode, e.g., the $inflow = outflow = 0$ constraint of the *Closed* mode of a valve. Each transition is either a nominal transition, modeling an executive command, or a failure transition.

Nominal transitions have preconditions that model the conditions under which that transition may be taken, e.g., in the absence of failure, a valve transitions from *Open* to *Closed* when it receives a *Close* command. At any given time, exactly one nominal transition is enabled, but zero or more failure transitions may be possible, e.g., a *Closed* valve may fail by transitioning to the *Stuck open* or *Stuck closed* modes. Hence, transitions have associated probabilities, which are used to model the likelihood of a failure occurring. Probabilistic failure transitions can be used to model intermittency, e.g., an *On* valve driver may fail by transitioning to the *Resettable failure* mode, but may transition back to *On* without any explicit command. Nominal transitions also have associated costs, providing a way to model the different costs of command sequences. For example, the least cost way of repairing a valve driver exhibiting *Resettable failure* is to *Reset* it, rather than turning it off and then on.

Components within a larger system can be viewed as acting concurrently, communicating over "wires." Hence, as before, system models are built compositionally by connecting

component transition system models. The resulting model is a *concurrent* transition system model in the sense that a single transition of the system corresponds to concurrent transitions by each of the component transition systems. Naturally, component transitions are consistent with the component connections. For example, the *Open/Close* command input to the valve is not directly controllable, but rather is an output from the valve driver. Hence, a valve transition can be commanded only if the valve driver is *On*.

We have built a model-based programming language that supports the specification of concurrent transition system models. This specification is compiled down into a restricted propositional temporal logic formula, which is used by Livingstone's MI and MR components. We have found that this modeling formalism has enabled us to naturally model (a) discrete, digital systems, e.g., the valve driver; (b) analog systems using qualitative modeling [46; 15], e.g., the valve; and (c) real-time software, e.g., the spacecraft attitude controller. Hence, the primary lesson of our experience is:

> *Concurrent transition systems provide an appropriate formalism for building model-based autonomous systems.*

5.1.2 Qualitative modeling

As noted above, we used qualitative representations for modeling analog systems. Sacks and Doyle [43] have strongly criticized the value of such qualitative representations, arguing that they can be used to analyze only a handful of simple systems. They conclude their critique with the comment that "[Qualitative] equations are far too general for practical use." [43].

Our experience in DS-1 has been quite to the contrary. We used extremely simple qualitative representations, primarily based on qualitative deviations from nominal, to model the analog systems of interest. We found that such representations were more than adequate for Livingstone's mode identification and reconfiguration tasks. Furthermore, the very simplicity of the models had important benefits. First, in contrast to detailed quantitative models, they are easy to acquire. We did not have to tease out the exact form of quantitative equations, or worry about carefully tuning numerical parameters. This enabled us to rapidly prototype the fault protection system concurrently with hardware design. Second, qualitative models provide a measure of robustness to design changes. For example, if the hardware designers choose to substitute a different thruster valve to produce more thrust, the qualitative model does not change: while the underlying meaning of nominal thrust changes, the qualitative model in terms of deviations from nominal remains the same. Third, qualitative models allow us to use propositional encodings that enable fast inference. This was essential to providing rapid and timely response. (We discuss this point in detail shortly.) The essential lesson we draw from our experience is the following:

> *Qualitative models are appropriate for many practical and significant tasks.*

5.2 Reactivity and deduction

A key contribution of Livingstone is that it unifies the dichotomy within AI between deduction and reactivity. Several authors, principally [1; 3], have argued that symbolic reasoning

methods, such as planning, deduction, and search, are unable to bridge the gap between perception and action in a timely fashion. For example, in discussing the construction of reactive systems that rapidly handle the complexity, uncertainty, and immediacy of real situations, Agre and Chapman claim that "Proving theorems is out of the question." [1]. Rather, the argument goes, the right way to construct reactive systems is to compile out all the inference into a network of combinational circuits, possibly augmented with timers and state elements, leading to the subsumption architecture [3]. But is this solution adequate for all types of reactive systems? More importantly, is the intuition, that deduction and search can play no role in reactive systems, even correct?

5.2.1 Fast deduction and search

Consider, first, the question of the adequacy of the above thesis. Autonomous system, such as deep space probes, Antarctic and Martian habitats, power and computer networks, chemical plants, and assembly lines, need to operate without interruption for long periods, often in harsh environments. In such systems, rapid correct response to anomalous situations is essential for carrying out the mission. Responding to any single anomalous situation using a hardwired network is plausible. However, as the length of time for which autonomous operations is desired increases, the combinations of anomalous situations that may arise grows exponentially. Constructing a reactive network that responds correctly to this cascade of failures is a truly daunting task. The model-based paradigm embodied in Livingstone, with its ability to identify multiple failures and synthesize correct responses directly from a compact declarative model, provides a much more practical solution.

But what of the concern that search and deduction are sufficiently time-consuming that responses at reactive time-scales are not possible? Livingstone addresses this concern with a combination of techniques (see [48] for details). We formulate both MI and MR as combinatorial optimization problems: MI is formulated as finding the most likely transitions that are consistent with the observations; MR is formulated as the least cost commands that restore the current configuration goals. Livingstone solves these combinatorial optimization problems using a *conflict-directed*[*] *best-first search*, coupled with fast propositional inference using *unit propagation*. Empirically, the use of conflicts dramatically focuses the search, enabling rapid diagnosis and response. While unit propagation is an incomplete inference procedure, it suffices for our applications. The reason is that we use causal models, with few (if any) feedback loops, so that unit propagation is complete or can be made complete with a small number of carefully chosen prime implicates [9].

5.2.2 Truth maintenance

Livingstone's performance is further significantly enhanced by using a *truth maintenance system*, called the *Incremental Truth Maintenance System* ITMS [35], which provides the propositional inference capability. The ITMS caches inferences during search, so that very little new inference needs to be done at each node in the search tree. The ITMS is a

[*]A conflict is a partial assignment such that any assignment containing the conflict is guaranteed to be infeasible.

variant of the more traditional LTMS [30; 16] that optimizes context switching. Context switching involves simultaneously deleting and adding clauses to the TMS database. The main drawback of the LTMS labeling algorithm is its conservative approach to context switching: all inferences that depend on a deleted clause are first removed, even if these inferences continue to hold in the new context. The ITMS takes a more aggressive approach to context switching by first adding clauses and propagating, and only then deleting clauses, thereby retaining significantly more inferences across the context switch.

Livingstone's use of an ITMS is in sharp contrast to other model-based diagnosis systems [13; 14; 10; 18] that use a fundamentally different type of truth maintenance system, called the ATMS [12]. Concerns about the efficiency of the LTMS lead de Kleer to introduce the ATMS and write that JTMSs and LTMSs "... have proven to be woefully inadequate... they are inefficient in both time and space." [8]. The advantage of the ATMS is its ability to switch contexts without any label propagation. However, this comes at the cost of an exponential time and space labeling process, making it inapplicable for embedded, real-time systems. This is not surprising since the original ATMS was designed for problems that require finding all solutions, e.g., envisionment, while Livingstone instead focuses on a small number of most preferred solutions. More recently, various ATMS focusing algorithms have been developed to alleviate the exponential cost of labeling by restricting ATMS label propagation to just the current context [21; 17]. Precise empirical comparisons between model-based diagnosis systems based on focused ATMSs and those based on LTMSs/ITMSs are unavailable. However, our experience with Livingstone on a standard diagnostic suite have been exceedingly favorable, and appear to be comparable to the very best focused ATMS-based diagnosis engines.

The primary lesson of the above discussion is the following:

> *Search and deduction are often necessary in a reactive system. Furthermore, search and deduction can be carried out reactively.*

5.3 Summary

Livingstone is a discrete model-based controller that provides the mode identification and reconfiguration capability within the Remote Agent architecture. Our experience with Livingstone has provided the following technical lessons:

- Multiple tasks can be carried out using a single model.
- Concurrrent transition systems provide an appropriate formalism for building model-based autonomous systems.
- Qualitative models are appropriate for many real-world tasks.
- Search and deduction are often necessary in a reactive system.
- Search and deduction can be carried out reactively.

6 Lesson from technology insertion

The strict separation between modeling and problem solving heuristics also addresses another lesson learned from the DS1 experience. While AI planning research has so far concentrated on problem-solving performance, in mission-critical applications it is *validation* of the problem-solving system that takes a much more prominent role. In our interaction with spacecraft engineers the question that is most often and insistently asked is "How can we be sure that your software will work as advertised and avoid unintended behavior?". Indeed, this is a question that applies to the development of all aspects of mission-critical embedded software systems, AI based or not. However, systems like Remote Agent promise complete autonomy over a much wider variety of complex situation than it was previously possible. On one hand this makes validation of these systems much harder than before. On the other hand, the declarative nature of AI technology allows the inspection of the models and the deep understanding of the behavior of the system in a way that is unprecedented with respect to traditional software development approaches.

A clean separation between models and heuristics allows the integration of AI technology to match much better to the realities of the development of large software systems. In such endeavours knowledge is distributed across people with different backgrounds and skills. In spacecfaft mission development, the knowledge of the behavior of the hardware and the specification of the procedures needed to achieve the mission goal resides in the *system and mission engineering* organization. Ultimately it is their responsibility to certify that hardware and software guarantee the mission requirements and specifications. The ability to directly inspect and understand actual flight software is enormously facilitated by the use of declarative methods. However, it is important to make sure that system and mission engineers can really focus on what they are primarily interested with (guaranteeing that requirements are met) and not on the details of how the reasoning engines need to manipulate the models in order to produce solutions efficiently. A strict separation between models and heuristics will allow non-AI specialists to understand the knowledge enbedded in the system without having to be experts in AI problem solving methods.

We believe that inspectable representational techniques and tools to automatically analyze models and automatically synthesize problem solving heuristicsis are important research areas to widen the applicability of AI techniques to real-world applications.

References

[1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 268–272, 1987.

[2] Douglas E. Bernard, Gregory A. Dorais, Chuck Fry, Edward B. Gamble Jr., Bob Kanefsky, James Kurien, William Millar, Nicola Muscettola, P. Pandurang Nayak, Barney Pell, Kanna Rajan, Nicolas Rouquette, Benjamin Smith, and Brian C. Williams. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference* [27].

[3] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[4] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[5] Peter Cheeseman, Bob Kanefsky, and William Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 163–169, 1991.

[6] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, 1989.

[7] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[8] Johan de Kleer. Choices without backtracking. In *Proceedings of AAAI-84*, pages 79–85, 1984.

[9] Johan de Kleer. Exploiting locality in a TMS. In *Proceedings of AAAI-90*, pages 264–271, 1990.

[10] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of AAAI-91*, pages 842–848, 1991. Reprinted in [25].

[11] Johan de Kleer, Alan Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992. Reprinted in [25].

[12] Johan de Kleer and Brian C. Williams. Reasoning about multiple faults. In *Proceedings Conference of the American Association for Artificial Intelligence*, pages 132–139, 1986.

[13] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. Reprinted in [25].

[14] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, pages 1324–1330, 1989. Reprinted in [25].

[15] Johan de Kleer and Brian C. Williams, editors. *Artificial Intelligence*, volume 51. Elsevier, 1991.

[16] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[17] Oskar Dressler and Adam Farquhar. Putting the problem solver back in the driver's seat: Contextual control of the ATMS. In *Lecture Notes in Artificial Intelligence 515*. Springer-Verlag, 1990.

[18] Oskar Dressler and Peter Struss. Model-based diagnosis with the default-based diagnosis engine: Effective control strategies that work in practice. In *Proceedings of ECAI-94*, 1994.

[19] R. James Firby. *Adaptive execution in complex dynamic worlds.* PhD thesis, Yale University, 1978.

[20] R. James Firby. The RAP language manual. Animate Agent Project Working Note AAP-6, University of Chicago, March 1995.

[21] Kenneth D. Forbus and Johan de Kleer. Focusing the ATMS. In *Proceedings of AAAI-88*, pages 193–198, 1988.

[22] Erann Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In Louise Pryor, editor, *Procs. of the AAAI Fall Symposium on Plan Execution.* AAAI Press, 1996.

[23] Erann Gat and Barney Pell. Abstract resource management in an unconstrained plan execution system. In *Proceedings of the IEEE Aerospace Conference* [27].

[24] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. Technical Report 411, Artificial Intelligence Center, SRI International, January 1987.

[25] Walter Hamscher, Luca Console, and Johan de Kleer. *Readings in Model-Based Diagnosis.* Morgan Kaufmann, San Mateo, CA, 1992.

[26] Walter C. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51:223–271, 1991.

[27] IEEE. *Proceedings of the IEEE Aerospace Conference*, Snowmass, CO, 1998.

[28] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[29] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[30] David McAllester. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Laboratory, August 1980.

[31] D. McDermott. A reactive plan language. Technical report, Computer Science Dept, Yale University, 1993.

[32] J. Muller and M. Pischel. An architecture for dynamically interacting agents. *Int. Journal of Intelligent and Cooperative Information Systems*, 3(1):25–45, 1994.

[33] Nicola Muscettola. HSTS: Integrating planning and scheduling. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.

[34] Nicola Muscettola, Paul Morris, Barney Pell, and Ben Smith. Issues in temporal reasoning for autonomous control systems. In Wooldridge [49]. To appear.

[35] P. Pandurang Nayak and Brian C. Williams. Fast context switching in real-time propositional reasoning. In *Proceedings of AAAI-97*, 1997.

[36] 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), March 1998.

[37] 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.

[38] Barney Pell, Gregory A. Dorais, Christian Plaunt, and Richard Washington. The remote agent executive: Capabilities to support integrated robotic agents. In Alan Schultz and David Kortenkamp, editors, *Procs. of the AAAI Spring Symp. on Integrated Robotic Architectures*, Palo Alto, CA, 1998. AAAI Press.

[39] 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. In Wooldridge [49]. To appear.

[40] Barney Pell, Erann Gat, Ron Keesing, and Nicola Muscettola. Plan execution for autonomous spacecraft. In *Proceedings of the 1996 AAAI Fall Symposium on Plan Execution*, 1996.

[41] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Robust periodic planning and execution for autonomous spacecraft. In *Procs. of IJCAI-97*, Los Altos, CA, 1997. IJCAI, Morgan Kaufman Publishers.

[42] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987. Reprinted in [25].

[43] Elisha P. Sacks and Jon Doyle. Prolegomena to any future qualitative physics. *Computational Intelligence*, 8(2):187–209, 1992.

[44] Bart Selman, David Mitchell, and Hector Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81:17–29, 1996.

[45] Peter Struss and Oskar Dressler. Physical negation: Integrating fault models into the General Diagnostic Engine. In *Proceedings of IJCAI-89*, pages 1318–1323, 1989. Reprinted in [25].

[46] Daniel S. Weld and Johan de Kleer, editors. *Readings in Qualitative Reasoning About Physical Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[47] Brian C. Williams and P. Pandurang Nayak. Immobile robots: AI in the new millennium. *AI Magazine*, 17(3):16–35, 1996.

[48] 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, AAAI Press.

[49] M. Wooldridge, editor. *Proceedings of the Second International Conference on Autonomous Agents*. ACM Press, 1998. To appear.