Remote Agent: To Boldly Go Where No AI System Has Gone Before*

Nicola Muscettola†
P. Pandurang Nayak‡
Barney Pell‡
Brian C. Williams
NASA Ames Research Center, MS 269-2,
Moffett Field, CA 94035.
Email: {mus,nayak,pell,williams}@ptolemy.arc.nasa.gov

Abstract

Renewed motives for space exploration have inspired NASA to work toward the goal of establishing a virtual presence in space, through heterogeneous fleets of robotic explorers. Information technology, and Artificial Intelligence in particular, will play a central role in this endeavor by endowing these explorers with a form of computational intelligence that we call *remote agents*. In this paper we describe the Remote Agent, a specific autonomous agent architecture based on the principles of model-based programming, on-board deduction and search, and goal-directed closed-loop commanding, that takes a significant step toward enabling this future. This architecture addresses the unique characteristics of the spacecraft domain that require highly reliable autonomous operations over long periods of time with tight deadlines, resource constraints, and concurrent activity among tightly coupled subsystems. The Remote Agent integrates constraint-based temporal planning and scheduling, robust multi-threaded execution, and model-based mode identification and reconfiguration. The demonstration of the integrated system as an on-board controller for Deep Space One, NASA's first New Millennium mission, is scheduled for a period of a week in late 1998. The development of the Remote Agent also provided the opportunity to reassess some of AI's conventional wisdom about the challenges of implementing embedded systems, tractable reasoning, and knowledge representation. We discuss these issues, and our often contrary experiences, throughout the paper.

Keywords: autonomous agents, architectures, constraint-based planning, scheduling, execution, reactive systems, diagnosis, recovery, model-based reasoning

*Authors in alphabetical order.
†Recom Technologies
‡RIACS

1 Introduction

The melding of space exploration and robotic intelligence has had an amazing hold on the public 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. In fact, 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 space station is far more modest in scope.

While this reality is far from our ambitious dreams for humans in space, space exploration is surprising 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

Providing a virtual human presence in the universe through the actual presence of a plethora of robotic probes 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 re-examine, 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 Presence 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, led 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" [47]. 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 missions, such as the Galileo

Figure 1: Planned and concept missions to extend human virtual presence in the universe. (1) Mars Sample Return missions (courtesy of NASA Johnson Space Center); (2) cryobot and hydrobot for Europa oceanographic exploration (courtesy of JPL); (3) DS3 formation flying optical interferometer (courtesy of JPL); (4) Mars solar airplane (courtesy of NASA Ames Research Center).

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 Pathfinder (MPF) mission introduced a paradigm shift within NASA towards lightweight, highly focused missions, at a tenth of the cost, and operated by small ground teams [14]. The viability of this concept was vividly demonstrated last summer when MPF landed on Mars and enabled the Sojourner micro-rover [48] to become the first mobile robot to land on the surface of another planet.

Pathfinder and Sojourner demonstrate an important mechanical means to achieving a virtual presence, but currently lack the on-board 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. 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 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 [16], such as the New Millennium Deep Space Three (DS3) mission. These interferometers 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 Galileo 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 on-board 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), has 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 re-*

source constraints. Tight deadlines that give no second chances stem from orbital dynamics and rare celestial events, and include examples such as executing an orbit insertion maneuver within a fixed time window, taking asteroid images during a narrow window around the time of closest approach, and imaging a comet's fiery descent into Jupiter. 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 Autonomy Architecture Prototype (NewMAAP), a simulation study based on the Cassini 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 mode identification and reconfiguration. An overview of the architecture is provided in Section 2. Additional details, including a description of the NewMAAP scenario, may be found in [57]. 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 are found in [5].

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" [18]
- "[Qualitative] equations are far too general for practical use." [63]

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 on-board *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 [70]. Model-based programming is based on the observation that programmers and operators generate the breadth of desired functionality from common-sense 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 mode identification and reconfiguration 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 3). 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 pre-computed. 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 pre-computing 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, surprisingly simple, deductive tasks were NP-hard. For example, after proving that his formulation of STRIPS-style planning was NP-hard, David Chapman concluded [11]:

> "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 [12, 64]. The surprisingly elusive hard problems lie at a phase transition from solvable to unsolvable problems. The elusiveness of hard problems, at least in the space of randomly generated problems, suggests that many real world problems may be tractable. This raises the possibility that a carefully designed and constrained deductive kernel could 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 [71, 56]. 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.

1.4.3 Goal-directed, closed loop commanding

A mission like Cassini requires a ground crew of 100 to 300 personnel at different mission stages. The driver for such a large team is not so much Cassini's nominal mission, but the effort required to robustly respond to extraordinary situations. Likewise, the need for extreme robustness without extensive ground interaction is Remote Agent's most defining requirement.

Traditional spacecraft are commanded through a time-stamped sequence of extremely low-level commands, such as "open valve-17 at 20:34 exactly." This low level of direct commanding with rigid time stamps leaves the spacecraft little flexibility when a failure occurs, so that it is unable to shift around the time of commanding or to change around what hardware is used to achieve commands.

A fundamental concept supporting robustness in classical control systems is feedback control. Feedback control avoids the brittleness of direct commanding by taking a set point trajectory as input and, using a feedback mechanism that senses the system's actual trajectory, commanding the system until the error between the actual and intended trajectories is eliminated. The set-point trajectory is a simple specification of an intended behavior, which gives the feedback controller freedom to determine the commands necessary to achieve this behavior

The Remote Agent embodies the same concept at a much more abstract level. It is commanded by a goal trajectory (the *mission profile*) that specifies high-level goals during different mission segments, such as performing an engine calibration activity within a 24 hour window before approaching the target. This gives the Remote Agent considerable flexibility as to how these goals are achieved. To achieve robustness, the Remote Agent uses its sensor information to continuously close the feedback loop at the goal level, quickly detecting and compensating for anomalies that cause the system to deviate from the goal trajectory.

Traditionally this feedback loop is closed by astronauts and the ground crew. A popular example that highlights the diverse actions humans can take to close this loop in extraordinary situations is the Apollo 13 crisis. The crisis began when a quintuple fault occurred, consisting of three electrical shorts, and a tank-line and a pressure jacket bursting. A first challenge for the ground crew was to accurately assess the health state of the spacecraft from its limited sensor information. No repair to the spacecraft would get the mission back on track to the moon, hence the second challenge involved quickly designing a new mission

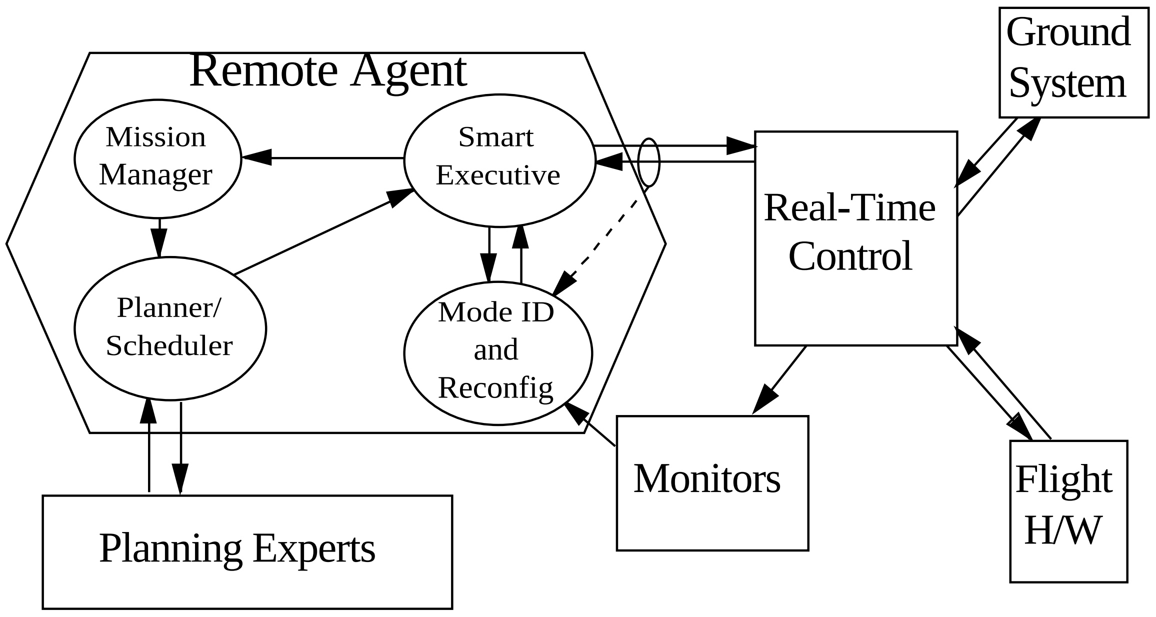

Figure 2: Remote Agent architecture embedded within flight software.

sequence that would allow the Apollo capsule to return to Earth in its hobbled state. Recall that astronaut Mattingly worked extensively in a ground simulator, in search of a novel command sequence that would work within the severe power limitations of the imperiled spacecraft. Ultimately Mattingly achieved this only through a novel, but unintended reconfiguration of the spacecraft hardware that drew current from the lunar module's battery. Finally, Astronauts Swaggert and Lovell had the challenge of quickly assembling together procedures that would guide the capsule through Mattingly's new mission sequence. This example highlights four basic roles performed by humans, that must also be embodied, albeit in a simpler form, within the Remote Agent. The first two roles, diagnosis of multiple failures and novel reconfiguration of hardware is performed by Remote Agent's model-based mode identification and reconfiguration component. Generation of new mission sequences under tight resource constraints is performed by the Remote Agent's planner/scheduler. Flexible assembly and execution of flight procedures to implement new and changing mission sequences is implemented by the Remote Agent's executive component.

In the next section we discuss how each of these components interact within the Remote Agent architecture. We then focus on technical lessons related to the three components of the Remote Agent, and then discuss key technology insertion lessons.

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 1.2. 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 execution time activity, 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 2. When viewed as a black-box, RA sends out commands to the *real-time control* system (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 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*.

Planner/Scheduler (PS) and Mission Manager (MM): PS is a constraint-based integrated temporal planner and resource scheduler [52] that 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 can be updated from the ground when necessary. It contains a list of all nominal goals to be achieved during the mission. For example, the DS1 mission profile contains goals such as optical navigation goals, which specify the duration and frequency of time windows within which the spacecraft must take asteroid images to be used for orbit determination by the on-board navigator. MM determines the goals that need to be achieved in the next horizon, e.g., a week or two 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 uses a heuristic guided backtrack search to produce 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. For example, a typical DS1 plan to achieve the above optical navigation goal requires the camera to be turned on and the spacecraft to be pointing at the asteroid before the image is taken. The plan leaves temporal flexibility on exactly when these events take place, and does not constrain the particular mode used by the attitude controller in effecting the turn.

Other on-board 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 main engine thrust goals based on its determination of spacecraft orbit, 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 [35], to define alternate methods for decomposing activities. For example, a high-level activity in DS1 such as thrusting the main engine is decomposed into coordinated commands to the main engine to start thrusting and to the attitude controller to switch into thrust vector control mode, and is executed only after the previous optical navigation window has ended.

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 recipient or by inferences drawn by the mode identification component of MIR. For example, when EXEC turns on the camera to prepare for imaging, MIR uses information from switch and current sensors to confirm that the camera did turn on. 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 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 [71], 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. For example, particular combinations of attitude errors allow MI to infer that a particular thruster has failed. EXEC is only informed about the failed state of the thruster, and not about the observed 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. Unlike PS, MR has a reactive focus, thus enabling it to rapidly suggest command sequences. 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. For example, if MI determines that the camera did not turn on when commanded, EXEC will request MR to repair the camera. MR will respond by instructing EXEC to retry the command.

3 Planning and scheduling

The Planner/Scheduler (PS) of the Remote Agent provides the high-level, deliberative planning component of the architecture. The *extended duration* of a space mission, coupled with the *unpredictability* of actions like thrusting, poses a challenge for planning, since it is impossible to plan the entire mission at the lowest level of detail. The approach in RA is to perform *periodic planning* [60], in which each round of planning has a restricted *scheduling horizon*. However, this raises a potential coherence problem, as activities within one horizon might compromise activities later in the mission (for example, aggressive maneuvers early in the mission may exhaust propellant needed for much later mission goals). RA addresses this problem through the Mission Manager (MM) component. When MM extracts goals for an upcoming round of planning, it also extracts constraints associated with the next *waypoint* in the mission profile. For example, a waypoint constraint specifies the amount of propellant that must be available for future use. By adding waypoint constraints to the current plan request, MM restricts PS to generate only plans that are coherent with the overall mission plan [61]. Hence, PS receives from MM and EXEC the initial spacecraft conditions, the goals for the next scheduling horizon, and the waypoint constraints. It produces a plan, which can be viewed as a high-level program that EXEC must follows in order to achieve the required goals.

Figure 3 shows the structure of PS (see [52, 51] for more details). A general-purpose *planning engine* provides a problem solving mechanism that can be reused in different application domains. A special-purpose *domain knowledge base* 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) framework. The search engine calls the plan database to record the consequences of each problem solving step and to require consistency maintenance and propagation services.

The search engine, Iterative Refinement Search (IRS), is a chronological backtracker that encodes a set of methods usable to extend a partial plan. 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*.

One crucial aspect of the success of PS is 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. The models expressed in DDL use two strong domain organizational principles that are the foundation of HSTS. First, it structures the description of the system as a finite set of *state variables*. A plan describes the evolution of a system as a set of parallel histories (timelines) over linear and continuous time, one per state variable. Second, it uses a unified representational primitive, the *token*, to describe both actions and state literals. As in [26], 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 principles in section 3.2.

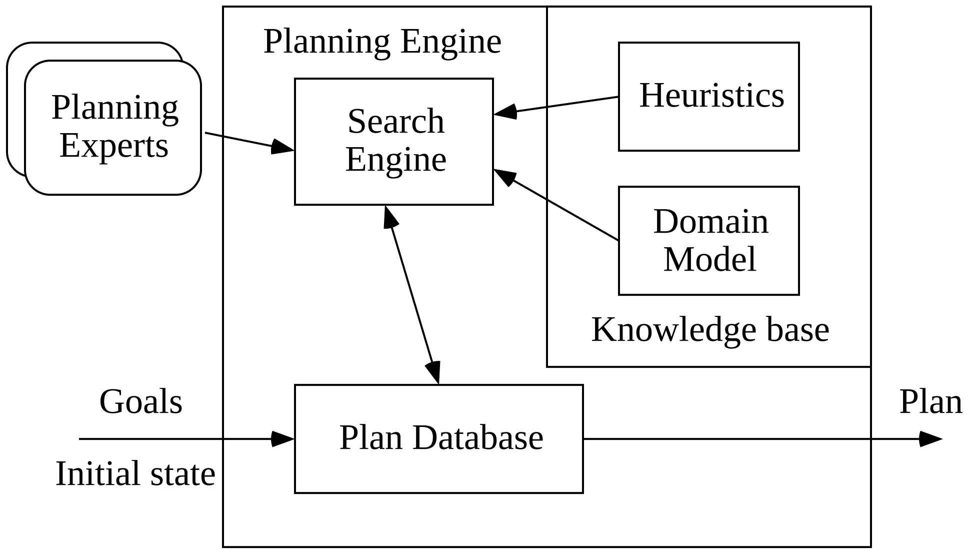

Figure 3: PS architecture diagram.

PS generates complex plans with performance acceptable for an on-board spacecraft application, even when using a very simple search strategy and a very simple heuristic language to program the search engine. This is because of 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 the structural principles of HSTS.

PS is a concrete example of the fact that, by solely relying on concepts and techniques from AI planning and scheduling research, it is possible to solve complex problems of practical significance. These techniques include subgoaling, temporal reasoning, constraint propagation, and heuristic search. Furthermore, we believe that at the current time AI planning and scheduling techniques provide the most viable software engineering approach to the development of high-level commanding software for highly autonomous systems. This bears great promise 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. The classical AI planning problem is to achieve 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 [32], which sees the world as an alternation of indefinitely persistent states and instantaneous actions. Classical schedulers, on the other hand, see the world as a set of resources and a set of structured task networks, with each task having a duration that is known a priori. 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 [51] 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 operational 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 where 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 PS decides the sequence of source and destination pointings within which the turn is inserted.

- *continuous parameters*: in addition to time, the planner must keep track of the status of other continuous parameters. These include the level of renewable resources like battery charge or data volume and of non-renewable resources like propellant. 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. In several cases sophisticated software modules are already available that effectively model subsystem behaviors and mission requirements. PS must be able to exchange information with these planning experts. For example, in DS1 PS makes use of a Navigation expert which manages the spacecraft trajectory. The Navigation expert is in charge of feeding PS with beacon asteroid observation goals to determine the trajectory error and with thrusting maneuver goals to correct the 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 according to a pre-defined Deep Space Network availability schedule"), 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").
- *accumulation goals*: these arise in the handling of continuous level 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 requirements with those of other goals.
- *default goals*: these 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 Principles

We mentioned that PS has two strong structural principles regarding how to represent domain models. We call them the *state variable principle* and the *token principle.* We now discuss both of these in more detail.

- *State variable principle:* 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 with 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 glance, structuring a model with a finite set of state variables could appear quite restrictive. However, on further analysis one can see that using this perspective is quite natural even 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). Moreover using state variables can be advantageous during problem solving. 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 (e.g., state variable constraints in Satplan [42] and mutex relations in Graphplan [6]).

- *Token principle:* 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 system state variables over time.

This structural principle challenges a fundamental tenet of classical planning: the dichotomy between actions and states. To illustrate why 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 as a function of the discrepancy and the maximum acceptable attitude 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`.

When 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. Thrusters are usually more active when the acceptable error in attitude is smaller. 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 represented 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)` as two different parameter settings for the ACS control system, conceptually best represented with state literals.

In this example the distinction between actions and states is not clear. Given the above observations, PS takes a radical view and gives the same status to all literals. 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 dichotomy, we use the neutral term *token* to refer to such temporally scoped assertions.

A domain model contains constraint patterns that have to be in every consistent plan. For example, Figure 4 gives the DDL construct representing the token conditions needed in a plan for the DS1 Microelectronics Integrated Camera And Spectrometer (MICAS) to take an image. This action is represented in the plan by the token

```
MICAS.actions_sv = Take_Image (?id, ?orientation, ?exp_time, ?settings)
```

meaning that the state variable `actions_sv` of the system component `MICAS` assumes a ground value matching the `Take_Image` predicate for the duration of the token. The constraint descriptor includes the specification of functional dependency between parameters of the token. In the example, the function `Compute_Image_Duration` computes the value of the token duration (special variable `?duration`) as a function of the value of the token arguments `?exp_time` and `?setting`. Finally, the descriptor includes temporal relations that have to be satisfied with other tokens in order for a plan to be consistent with the domain model. In the example these constraints follow the `:temporal_relations` keyword. They state that the `MICAS.actions_sv` state variable must be `Idle` immediately before and after the `Take_Image` token; that `Take_Image` consumes 140 watts of power; that during `Take_Image` the spacecraft must be `Constant_Pointing` in the requested `?orientation`; and that during `Take_Image` `MICAS` must be both in good health (`MICAS_Available` token) and `Ready` for use.

The above constraint template is closely related to temporally scoped operators used in temporal planning approaches [3]. However, as a consequence of the token principle, our framework allows the expression of similar constraint patterns for "state" tokens like `MICAS.actions_sv = Idle`. In reality it is equally important to be able to express constraints both on "actions" and on "states". For example, a functional duration constraint may need to apply both to `Turning (?x, ?y)`, where duration depends on the angle between `?x` and `?y`), and to `Constant_Pointing (?z)`, where the maximum duration may depend on how the relative orientation of the Sun with respect to `?z` affects the satisfaction of solar exposure constraints for sensitive subsystems.

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.

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 [27]. 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

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

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

Figure 4: Taking a picture with the on-board MICAS camera.

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 posts portions of a constraint network in the plan database. The plan database then enforces consistency checking by propagating 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 propagate constraints at run time. EXEC's constraint propagation has very different requirements from that of PS (see Section 4.2.1).

3.4 Practical generative planning

Figure 5 outlines the PS search process. If the partial plan in the plan database has "flaws", PS 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

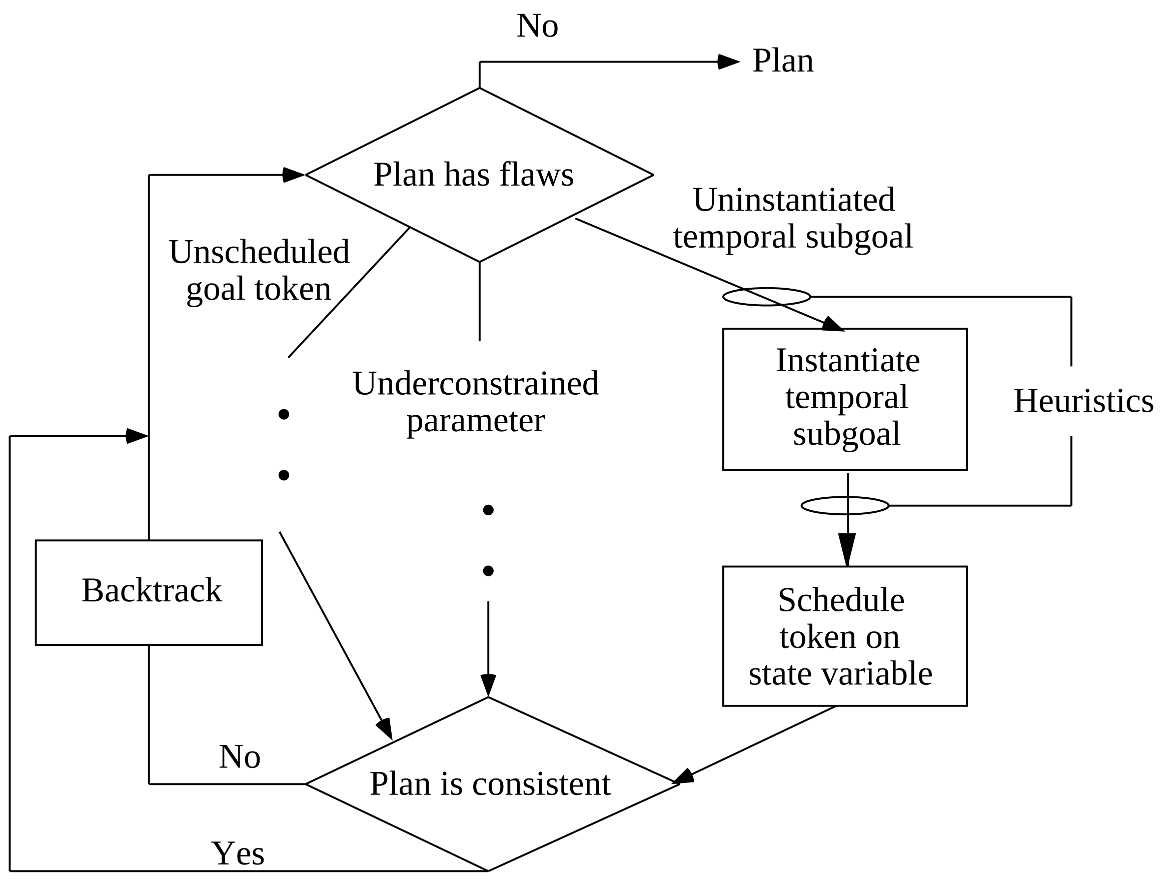
No
Plan
Plan has flaws
Unscheduled goal token
Uninstantiated temporal subgoal
Underconstrained parameter
Instantiate temporal subgoal
Heuristics
Backtrack
Schedule token on state variable
No
Plan is consistent
Yes

Figure 5: PS problem solving cycle.

ranges. If propagation detects an inconsistency, then PS chronologically backtracks. When the plan database contains no more flaws, a plan is returned.

The flaw detection and repair process is analogous to other classical planning algorithms [67]. PS recognizes several kind of flaws. Figure 5 lists three of them. The *uninstantiated temporal subgoal* flaw refers to a single temporal relation in a token compatibility and is resolved analogously to *open precondition* flaws in classical planning. *Unscheduled goal token* flaws refer to goal tokens for which a legal position on a state variable has not yet been found. PS resolves this flaw either by finding such a legal position or, if such a position cannot be found, by rejecting the goal. The *underconstrained variable value* flaw is handled by restricting the value range for a variable to a subrange (possibly a single value) of the original range. The handling of this flaw is analogous to *value selection* in constraint satisfaction search.

The prioritization of open flaws and the selection of alternatives during flaw handling relies on very simple heuristics. For example, uninstantiated temporal compatibilities are assigned a numeric priority according to the value range of the variables involved in the flaw at the moment the flaw first appears in the plan.

Although the search strategy and heuristic language are rather simple, PS can solve problems of size and complexity adequate for practical application domains. For example, the

DS1 Remote Agent experiment domain consists of 18 state variables, 42 token predicates and 46 compatibility specifications. The largest plan in the nominal Remote Agent experiment scenario has 154 tokens and 180 temporal constraints between tokens. This translates into an underlying constraint network with 288 variables and 232 constraints. Of these, 81 variables and 114 temporal-bound constraints constitute a simple temporal subnetwork that relates start and end times of tokens. The constraints in the rest of the network have an average arity (i.e., number of variables related by one constraint) of 3.5. The number of nodes expanded during plan generation is 649 with a search efficiency of about 64%. (Search efficiency is measured by the ratio of the number of nodes on the path to the solution and the total number of expanded nodes. Thus a search efficiency of 100% indicates no backtracking.)

From the previous description we can conclude that PS is a purely *generative* planner that operates at a single abstraction level. Most importantly, PS does not use pre-compiled plan fragments but assembles the overall plan from atomic components. This differentiates PS from most practical applications of planning technology to date [15, 69, 13]. These systems rely on Hierarchical Task Network (HTN) planning, in which most of the power comes from hand-generated task networks that are patched together into an overall plan. The notable absence of generative planning in successful applications has led to the commonly shared view that only HTN planning has true utility with respect to the automatic solution of planning problems of commercial significance.

Although pre-compiling token networks into HTN can be a powerful problem solving technique, our choice of pure generative planning is not accidental. First, not all domains are equally amenable to the HTN approach. For example, in DS1 the task decomposition hierarchy is shallow and useful pre-compiled task networks assemble only a small number of tokens. In these conditions the pre-compiled task networks do not significantly differ from the domain compatibilities and therefore HTN has no clear advantage with respect to generative planning. Second, replanning, especially with degraded capabilities, relies on representation of domain-level constraints between activities and goals, which are often not included in HTN representations [31]. Third, 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 independent 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. Our approach instead clearly separates between domain model and heuristics. As we shall see in section 6.2, the separation between domain models and problem solving heuristics is crucial to facilitate validation and has a big impact on the acceptability of AI technologies for mission-critical applications.

3.5 Summary

PS is a constraint-based, temporal planner that provides the high-level commanding capability for the Remote Agent architecture. From our experience we take the following lessons:

- Classical planning and classical scheduling must be combined and augmented for autonomous commanding of complex systems.

- The classical action/state dichotomy is problematic and should be substituted by the unified concept of a token.
- A constraint-based plan representation organized across state variables is a powerful problem-solving framework for planning.
- Heuristic generative planning can solve problems of practical significance.
- The separation between domain models and problem solving heuristics is important for validating planners in real-world domains.

4 Executive

EXEC is a robust event-driven and goal-oriented multi-threaded execution system. It provides a language and a framework in which software designers can express how planning, control, diagnosis, and reconfiguration capabilities are to be integrated into an autonomous system. It can request and execute plans involving concurrent activities that may be interdependent, where the success, timing, and outcomes of these activities may be uncertain. It provides a language for expressing goal-decompositions and resource interactions. When interpreting this language at run time, the executive automates the decomposition of goals into smaller activities that can be executed concurrently. This automates aspects of the labor-intensive sequencing function in spacecraft operations and raises the level of abstraction at which the ground system or on-board planner must reason. EXEC's design also supports a close integration between activity decomposition and fault responses. This leads to more robust execution, avoids loss of mission objectives, improves mission reliability and resource utilization, and simplifies the design of the entire software system.

EXEC is built on Execution Support Language (ESL) [35], which provides sophisticated control constructs such as loops, parallel activity, synchronization, error handling, and property locks [36]. These language features are used in EXEC to implement robust schedule execution, hierarchical task decomposition, context-dependent method selection, routine configuration management, and event-driven responses [58].

In the RA architecture, EXEC plays the main coordination role as the intermediary between the other flight software modules, both internal and external to RA. Here we concentrate 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 being executed. 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.

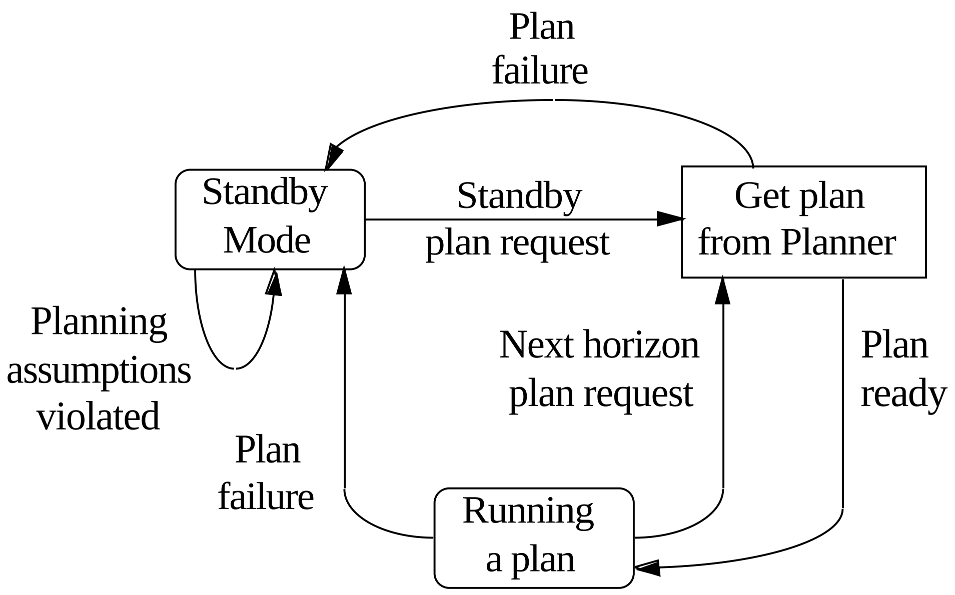

Figure 6: Periodic Planning and Replanning Cycle.

4.1 Periodic Planning over Extended Missions

Figure 6 shows both major branches of the periodic planning and replanning cycle in RA: *nominal execution* and *plan failure execution*. Nominal execution occurs if all plan tokens execute without MIR or EXEC detecting any execution failure. In this case plan execution proceeds to the end of the current scheduling horizon. At the pre-defined point in the plan, EXEC invokes PS, continues executing while waiting for the new plan and then smoothly installs the new plan into the current execution context.

Plan execution failure occurs if MIR or EXEC encounter an unrecoverable failure. In this event, EXEC aborts all current activity and enters standby mode, which serves (by design) as a well-defined invocation point for planning. EXEC then requests a new plan from this state (possibly updating the planner about degraded capabilities) and starts executing the plan as soon as it receives it back from PS.

A smooth execution of the periodic planning cycle requires EXEC to coordinate the generation of a new plan with other activities and to communicate information about available system capabilities to PS via MM.

4.1.1 Planning to plan

In the spacecraft domain, planning itself has informational preconditions (since planning relies on input planning experts, which 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. Domain constraints enforced in the plan ensure that their resources and preconditions will all be achieved be-

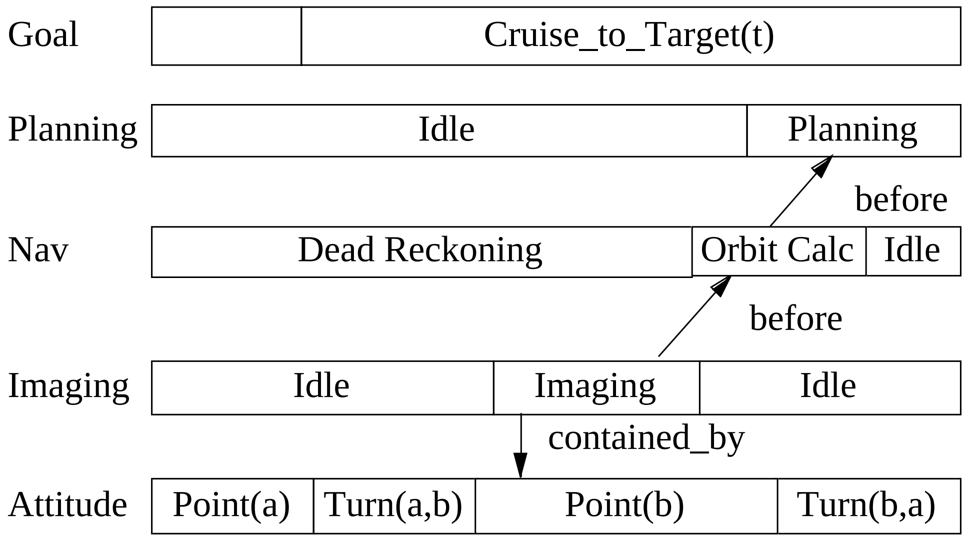

Figure 7: Sample Plan Fragment.

fore planning is invoked. This aspect of *planning to plan* [60] can be considered a form of meta-planning [7].

RA's approach to planning to plan is illustrated in Figure 7. In this example, the plan fragment has a constraint that the next round of planning should occur only after the Navigation expert 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` token 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 operate 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 adds 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 operate with high duty cycles in excess of 92% of the available time. Since other activities already require the IPS engine to be off (such as scientific experiments and observations), omitting IPS thrusting during planning would leave insufficient total thrust accumulation to reach the target. Hence, EXEC continues plan execution while PS is planning [60]. This necessitates tracking changes to the planning assumptions while planning, and using the currently executing plan for prediction about activities that will happen while planning for the next period is underway.

4.1.3 Replanning with degraded capabilities

When operating over extended periods of time, a spacecraft will 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 (see Section 5), EXEC will keep track of such degradation when commanding future planning cycles. For example, one fault mode in DS1 is for one of the 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 PS.

4.2 Robust Plan Execution

We have seen that in nominal operations EXEC invokes the planning machinery as a by-product of plan execution, which ensures that resources are available for planning and that the projected state used as a basis for planning is well-defined. 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 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 standby mode. This serves (by design) as a well-defined invocation point for planning.

Entering a standby mode following plan failure is costly with respect to mission goals, because it interrupts the ongoing planned activities and important mission opportunities may be lost. For example, a plan failure causing EXEC to enter a standby mode during a comet flyby would cause loss of all the encounter science, as there would be no time to re-plan before the comet passed 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 by:

- Executing flexible plans by running multiple parallel threads and using fast constraint propagation algorithms in EXEC to exploit plan flexibility.
- Choosing a high level of abstraction for planned activities so as to delegate as many detailed activity decisions as possible to the procedural executive.
- Handling execution failures using a combination of robust procedures and deductive repair planning.

4.2.1 Executing flexible plans

As discussed in Section 3.3, plans are constraint networks representing envelopes of desirable behaviors for the system. EXEC incrementally interprets plans and in doing so determines at run time the actual behavior of the system. This process involves propagating execution time information through the plan's constraint networks.

The process of interpreting a plan is carried out by EXEC's *plan runner*. Here is a brief sketch of how the plan runner works. The plan runner treats each state variable as a separate thread of execution. Each token on the state variable corresponds to a *program* that runs on that thread. The transition between one token and the following one on the state variable is represented by a *time point*, i.e., a time variable in the underlying plan constraint network. Starting and terminating the execution of tokens involves a certain amount of processing, which must be done for each time point in the plan. The plan runner has to wait until a time point is enabled, i.e., all time points that must precede it have been executed, and the current time is within its time bound. When a time point can be executed, the plan runner executes the following cycle:

1. Set the execution time of the time point to be the current time;
2. Set all parameters of the tokens started by the time point to one of the acceptable values;
3. Propagate the consequences of the previous value assignments to the rest of the plan;
4. Terminate execution of all tokens ending with the time point;
5. Start execution of all tokens starting with the time point;

After the execution of the previous cycle the plan runner waits until the current time enters the time bound of some other enabled time point. Note that step 3 adjusts the set of possible values for both the start/end time bounds and the parameters of still unexecuted tokens.

Making the plan runner work in a real-world application raised an issue that is often overlooked in AI execution research: the need for EXEC to give real-time guarantees about its operation.[1] A real-time guarantee can be seen as a way to quantify the "reactivity" of an agent. The way the problem arises in the plan runner is the following. Processing the execution cycle takes time, the *execution latency* λ. One can show that EXEC will be unable to guarantee the exact execution time of a time point with a precision finer than the latency. In other words, when EXEC is asked to execute an event precisely at time t, it can only guarantee that the actual event execution time will be somewhere in the interval $[t - \frac{1}{2}\lambda, t + \frac{1}{2}\lambda]$ [53]. Therefore, in order to produce a highly reactive and temporally precise agent (e.g., one that can guarantee taking a picture of a target within a few milliseconds of a given time) it is necessary to reduce the execution latency to a minimum.

One important way to reduce λ is to speed up the execution-time constraint propagation. In RA we address this speed-up problem for propagation in the time constraint network. One can show that it is always possible to transform a simple temporal constraint network into an equivalent one (i.e., one that represents the same set of consistent time assignment to time points) that is *minimal dispatchable*. Dispatchable means that EXEC's time propagation need only propagate execution time to the immediately adjacent time points in the temporal constraint network. Minimal means that the network contains the minimum number of edges among all dispatchable networks [54, 66]. This means that the flexible temporal constraint networks that PS gives to EXEC are those for which execution can be the fastest possible.

[1]An alternative approach to real-time execution guarantees is addressed in the CIRCA architecture [55].

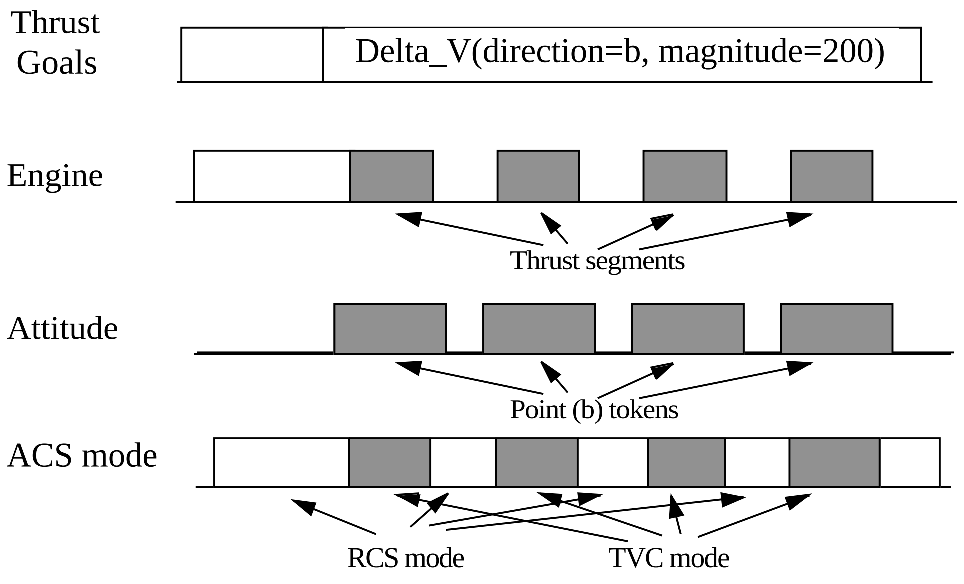

Figure 8: Plan fragment for achieving a change in spacecraft velocity.

4.2.2 Delegating activity details to execution

Generation of plans with temporal flexibility follows from the PS being 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). For each token type 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 DS1 illustrates this approach (see Figure 8). A `Delta_V` goal token requires the achievement of a certain change (delta) in the velocity of the spacecraft. Velocity changes are achieved by thrusting the engine for some amount of time while pointing the spacecraft in a certain direction. A total velocity change is achieved via a series of shorter thrust (time) 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.[2]

Initiating a thrust activity involves performing a number of complex operations on the engine and there is considerable uncertainty about how long this initiation takes before thrust starts accumulating. This translates into uncertainty about when to switch attitude control modes, how much thrust will be actually accumulated in a given segment, and how many

[2]In TVC mode, the spacecraft is turned by steering the main engine gimbal, whereas in ACS mode the spacecraft is turned using small attitude thrusters.

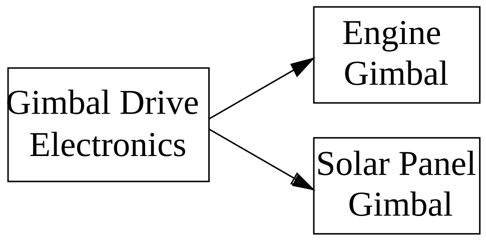

Figure 9: Interacting gimbal subsystems in DS1.

thrust segments are necessary to achieve the total desired thrust. RA takes 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 PS. 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. We note that this delegation of labor from PS to EXEC relies on many of the capabilities of a sophisticated procedural execution system [58, 36, 35].

4.2.3 Hybrid procedural/deductive executive

The preceding discussion has described ways to achieve robust execution primarily in the presence of uncertainty in timing or task progress. Another major cause of execution failure in the spacecraft domain is activity failure, often due to problems with the hardware. Several properties of the spacecraft domain drove us to design an executive that combines a rich procedural execution language with local recovery planning. These challenging properties include *tight coupling between subsystems*, *irreversible actions*, and *complex internal structure*.

As an example of tight coupling between subsystems, consider two spacecraft subsystems in DS1 (see Figure 9): the engine gimbal and the solar panel gimbal. A gimbal enables the engine nozzle to be rotated to point in various directions without changing the spacecraft orientation. The solar panels can be independently rotated to track the sun. In DS1, both sets of gimbals communicate with the main computer via a common gimbal drive electronics (GDE) board. If either system experiences a communications failure, one way to reset the system is to power-cycle (turn on and off) the GDE. However, resetting the GDE to fix one system also resets the communication to the other system. In particular, resetting the engine gimbal, to fix an engine problem, causes temporary loss of control of the solar panels. Thus, fixing one problem can cause new problems. To avoid this, the recovery system needs to take into account global constraints from the nominal schedule execution, rather than just making local fixes in an incremental fashion, and the recovery itself may be a sophisticated plan involving operations on many subsystems.

Another problem stems from the need to repair systems with complex internal structure and irreversible actions. For example, the propulsion system on the Cassini spacecraft [10]

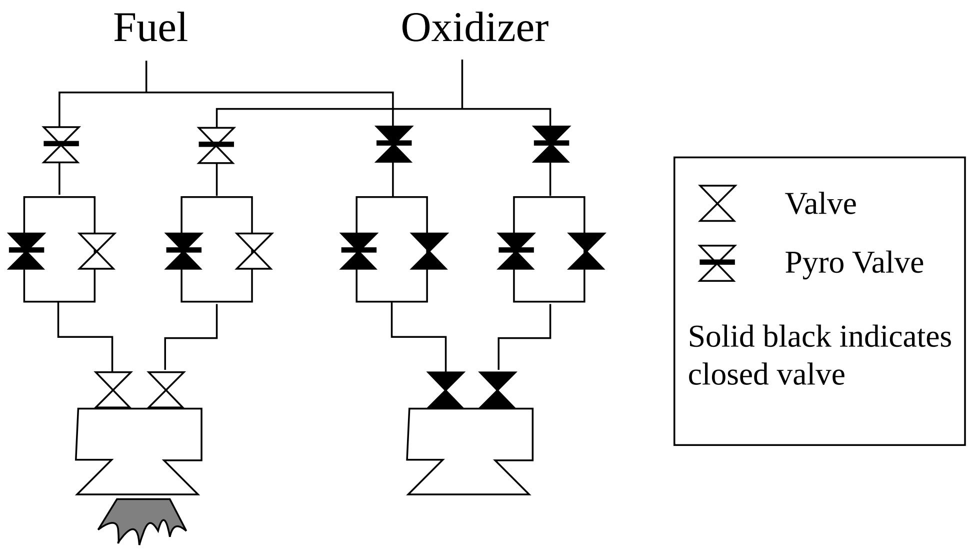

Figure 10: Simplified schematic of Cassini spacecraft propulsion system.

has a complex set of valves (see Figure 10) including explosive *pyro* valves, which can change states only once, and ordinary valves with varying amounts of wear and tear. It is difficult to express the right valve choices to redirect fluid flow while minimizing costs and risks in the wide variety of situations that might be encountered in flight.

Examples like these drove the design of RA's hybrid execution system [59], which integrates EXEC, the procedural executive based on generic procedures, with MIR, a deductive model-based executive (see Section 5) that provides algorithms for sophisticated state inference and optimal failure recovery planning.

RA's integrated executive enables designers to encode 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 ensures both that recovery sequences are consistent with high-level schedule execution and that a high degree of reactivity is retained to effectively handle additional failures during recovery.

The need to integrate EXEC with the local-recovery planning ability of MIR had a significant impact on the design of EXEC. In particular, we found that our integration approach required synchronization constructs that were not present in most execution languages. In the NewMAAP RA prototype, EXEC was based on the language provided by RAPS [33]. RAPS supports robust execution through the definition of multiple *methods* for each procedure. If one method fails, the RAP interpreter selects among alternate methods or variable bindings until it has run out of options, in which case the entire procedure fails. Effectively, RAPS handles failures on an activity-by-activity basis. The NewMAAP RA prototype followed a similar approach and EXEC invoked MIR to plan a recovery for each activity separately. However, the design of real flight software for DS1 introduced the problems of tightly interacting subsystems described above. This caused us to re-design the interface so that EXEC would suspend failed activities and provide global constraints (to preserve the health of functioning subsystems) as part of a request to repair failures. This turned out to be extremely difficult to do in RAPS, for three reasons. First, RAPS has no constructs for tasks to describe properties they need maintained for successful execution. Second, RAPS does not support nested contexts for recovery procedures, so that tasks can respond to failures in a specific way but ultimately draw on more generic recoveries. Third, RAPS does not support suspending threads based on external interrupts while a global recovery is in progress.[3] These difficulties motivated the design the new execution language, ESL, with facilities for easy language extension, a more flexible notion of concurrent activity and interrupts, hierarchical recovery procedures, and declarations of required properties [35, 36].

4.3 Summary

EXEC is a robust event-driven and goal-oriented multi-threaded execution system that coordinates the activity of the other flight software modules, both internal and external to the Remote Agent. This section has discussed the following major points:

- Coherent autonomous operation across a long-term mission can be achieved through periodic planning guided by a mission profile.
- Executing flexible constraint-based plans with bounded execution-time propagation results in robust plan execution with guaranteed real-time behavior.
- Procedural and deductive capabilities can be integrated within the reactive execution loop.
- Enhanced synchronization primitives to track state requirements are necessary for concurrent execution systems.
- A robust multi-threaded executive provides the core capabilities to support an architecture for autonomous operations over extended missions.

5 Model-based mode identification and reconfiguration

The mode identification and reconfiguration component of the Remote Agent architecture is provided by the Livingstone system [71]. Livingstone is a discrete model-based controller that sits at the nexus between the high-level feed-forward reasoning of classical planning and scheduling systems, and the low-level feedback control of continuous adaptive methods (see Figure 11). 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

[3]Similar concerns apply to other procedural execution systems, like PRS [37], RPL [46], Interrap [50] and Golog [43].

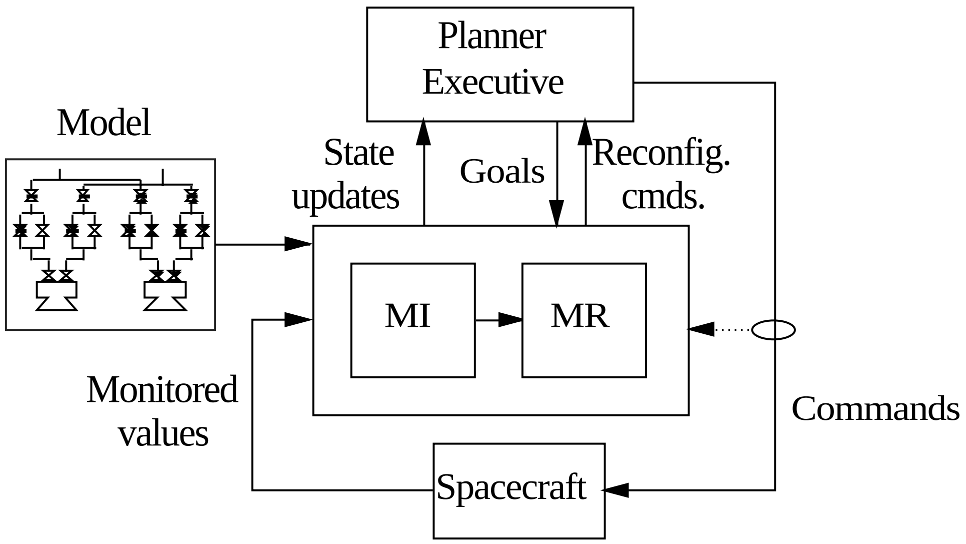

Figure 11: Livingstone architecture diagram.

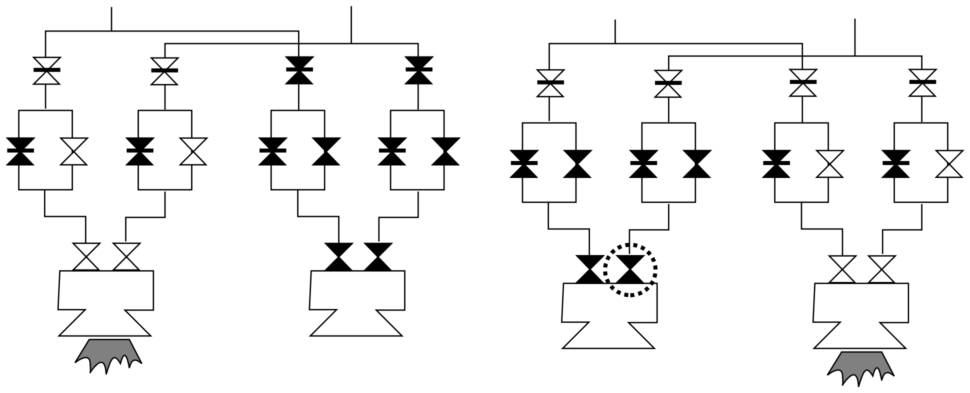

Figure 12: Different configurations that achieve thrust. The circled valve has failed.

to line and functional redundancy. For example, Figure 12 shows two configurations that satisfy the goal of providing thrust, with the one on the right being used when the circled valve fails. Other configurations, corresponding to different combinations of open valves, are used to handle other valve failures.

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 the fault and diagnoses its cause, thus identifying the actual spacecraft configuration, using algorithms adapted from model-based diagnosis [23, 24].

MI provides a variety of functions within the RA architecture, including:

- Mode confirmation: Provides confirmation to EXEC that a particular spacecraft command has completed successfully.

- Anomaly detection: Identifies observed spacecraft behavior that is inconsistent with its expected behavior.
- Fault isolation and diagnosis: Identifies components whose failures explain detected anomalies. In cases where models of component failure exist, identifies the particular failure modes of components that explain anomalies.
- Token tracking: Monitors the state of properties of interest to the executive, allowing it to monitor plan execution.

When the current configuration ceases to satisfy the active configuration goals, Livingstone's mode reconfiguration (MR) capability can identify a least cost set of control procedures that, when invoked, take the spacecraft into a new configuration that satisfies the goals. MR can be used to support a variety of functions, including:

- Mode configuration: Place the spacecraft in a least cost configuration that exhibits a desired behavior.
- Recovery: Move the spacecraft from a failure state to one that restores a desired function, either by repairing failed components or finding alternate ways of achieving the goals.
- Standby and safing: In the absence of full recovery, place the spacecraft in a safe state while awaiting additional guidance from the high-level planner or ground operations team.

Within the RA architecture, MR is used primarily to assist EXEC in generating recovery procedures, in response to failures identified by MI. (Section 4.2.3 has a more detailed discussion.)

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 [44] with the qualitative representations developed in model-based reasoning [68, 25]. Third, the approach unifies the dichotomy within AI between deduction and reactivity [1, 9], 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

Implemented model-based diagnosis systems traditionally specify behavior through constraint-based modeling, for example, see [17, 23, 65, 39]. In this formalism, 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.

While compositional, constraint-based modeling is well suited for many model-based diagnosis applications it has a significant limitation. Its widespread use is restricted by the fact that it typically has no model of *dynamics*, in other words, no model of transitions between modes. Modeling dynamics is essential for Livingstone since it needs to track changes in spacecraft configurations and determine reconfiguration sequences.

Most formalizations of model-based diagnosis, on the other hand, have assumed that models are specified in first-order logic [62, 21]. The enticement of first-order logic is its clear, well understood semantics. However, first order logic is not an accurate reflection of existing implementations, and is wholly inappropriate as a representation formalism for building practical diagnosis systems. On the one hand, its expressiveness 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 does not offer a particularly natural language for describing the dynamics of state change. Modeling dynamics is essential to modeling most hardware and software systems. Finally, first order logic by itself, is an impractical language for writing large scale models, with its flat structure of constants, functions, and relations.

Our challenge with Livingstone then was to develop a practical modeling language that is effective for compositional modeling, can represent the dynamics of hardware and software naturally, has a clean underlying semantics, and can be computed with efficiently in real-time.

5.1.1 Concurrent transition systems

We overcame the above limitation by coupling compositional constraint-based modeling with the *concurrent transition systems* used to model reactive software [44]. 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 13 shows the modes and transitions of a valve and a valve driver. Each transition is either a nominal transition, modeling an executive command, or a failure transition. As before, each mode is associated with a set of constraints that describe the component's behavior in that mode, for example, the $inflow = outflow = 0$ constraint of the *Closed* mode of a valve. To ensure that the representation is computable and has a well defined semantics, we restrict constraints to finite domains, and compile them down into propositional logic.

In terms of dynamics, nominal transitions have preconditions that model the conditions

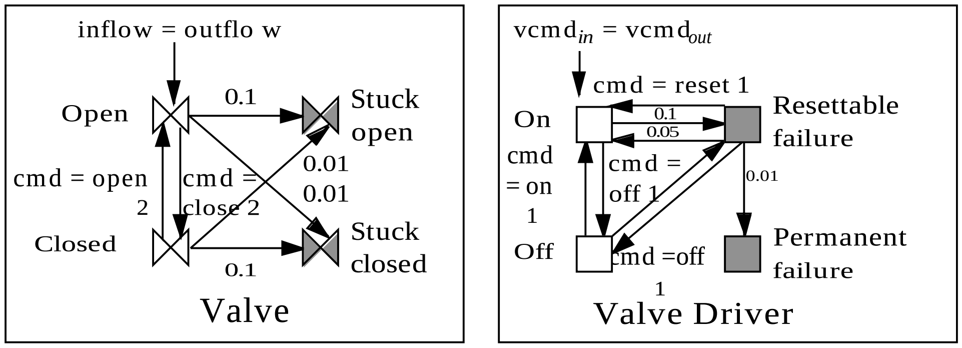

Figure 13: Transition systems for a valve and a valve driver. Shaded modes are failure modes. Fractional numbers represent transition probabilities and whole numbers represent transition costs.

under which that transition may be taken. For example, 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, for example, 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 intermittence, 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 with constraint-based modeling, 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*.

To support large scale modeling, we have built a compositional, model-based programming language that supports the specification of these concurrent transition system models. This specification is compiled down into a restricted propositional temporal logic formula with a well-defined semantics. This formula is used directly 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 [68, 25], e.g., the valve; and (c) real-time software, e.g., the spacecraft attitude controller. Hence, the primary lesson of our experience is:

Probabilistic, concurrent transition systems provide an appropriate formalism for *building model-based autonomous systems that is expressive, has a clean semantics, and is tractable.*

5.1.2 Qualitative modeling

As noted above, we use simple, qualitative representations for modeling analog systems. Sacks and Doyle [63] have strongly criticized the value of such qualitative representations, arguing that they are too ambiguous, and 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." [63]. Indeed much of the work on qualitative reasoning and model-based diagnosis has focused on a variety of methods that try to eliminate the computation of ambiguous values, by applying more and more quantitative information.

Our experience has been quite to the contrary. First, the fact that a model may lead to ambiguous values is no indicator of whether or not a representation is sufficient. In the case of diagnosis it simply must be the case that enough values have sufficient precision to rule out incorrect diagnoses. Second, the detail of modeling information necessary to rule out incorrect diagnoses can be very little. For example, researchers at Xerox PARC tried to develop a simplest set of copier models sufficient to cover all diagnoses listed within a human generated diagnostic repair procedure [4]. What they found is that the representations used in these models were far simpler than the representations Sacks and Doyle asserted were so impoverished.

Based on this lesson, we adopted a modeling formalism for Livingstone that models analog behaviors using an extremely simple representation based on qualitative deviations from nominal behavior. We found that such representations were more than adequate for all of 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, and can be acquired during early stages of the design process. 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 and modeling inaccuracy. 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:

Extremely simple, qualitative models are appropriate for many practical and significant tasks.

5.2 Reactivity and deduction

A key contribution of Livingstone is the fact that it unifies the dichotomy within AI between deduction and reactivity. Several authors, principally [1, 9], 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, for example, to the subsumption architecture [9]. But is this solution adequate for all types of reactive systems, particularly remote agents? Equally important, is it even correct that deduction and search can play no role in reactive systems?

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 compositional, 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 [71] 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 does follow the spirit, proposed by Brooks of compiling the system into a simple network. However, instead of a functional network that is evaluated, Livingstone compiles the models into a propositional constraint network. This is a very simple deductive search problem that we highly tune for performance. Our motive for reducing all model-based tasks to a highly tuned, search algorithm on propositional constraints parallels the intuitions behind reduced instruction set computers (RISC).

Livingstone solves these combinatorial optimization problems using a *conflict-directed*[4] *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 [19].

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

5.2.2 Truth maintenance

The above techniques allow Livingstone to identify modes and reconfigure hardware while evaluating only an extremely small set of candidate solutions. Hence the potentially exponential search appears not to be a major part of the problem. Nevertheless, with requirements for response times on the order of hundreds of milliseconds on a slow processor, unit propagation becomes a significant problem. Livingstone's performance is enhanced by another order of magnitude using a *truth maintenance system*, called the *Incremental Truth Maintenance System* ITMS [56], that computes unit propagations over time. The ITMS is a variant of the more traditional Logic-based TMS (LTMS) [45, 28] that optimizes context switching. The ITMS, like the traditional LTMS, computes truth assignments over a trajectory of states in an "event driven manner." That is, the ITMS propagates *changes* in truth assignments from one state to another, rather than performing a full unit propagation in every state.

Livingstone's use of an ITMS is in sharp contrast to other model-based diagnosis systems [23, 24, 20, 30] that use a fundamentally different type of TMS, called the Assumption-based TMS (ATMS) [22]. Concerns about the efficiency of the LTMS in the 80's lead de Kleer to introduce the ATMS and write that traditional TMSs "...have proven to be woefully inadequate...they are inefficient in both time and space" [18]. The advantage of the ATMS is its ability to switch contexts without any label propagation, thus avoiding the linear time cost of unit propagation. However, this comes at the cost of an exponential time and space pre-labeling process. These costs can be particularly problematic for embedded, real-time systems. 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 [24, 34, 29]. Precise empirical comparisons between model-based diagnosis systems based on focused ATMSs and those based on LTMSs/ITMSs are unavailable. However we did perform limited experiments on a version of Livingstone that contained *no* TMS. Even without a TMS, Livingstone's run time on a standard diagnostic test suite was comparable to diagnostic algorithms, such as Sherlock [24], that contain an ATMS.

This result led us to revisit the LTMS technology, which had received little attention over the last decade. We found that the addition of a traditional LTMS significantly improved Livingstone's performance. On the other hand, we also found that the LTMS performance was in some cases far from ideal. In the best case an LTMS update would be linear in the number of labels that change between successive states. Unfortunately, applying Livingstone to the DS1 spacecraft models, we found that the LTMS spent a significant percentage of its time on labels that remain constant; more specifically, 37% on average, and 670% in the worst case. This worst case can be deadly for real systems with hard time requirements. The ITMS offers a more aggressive approach to label update that is merely 5% off ideal, with a worst case overhead of only 100%. The use of this TMS resulted in an order of magnitude improvement in Livingstone's performance, over the version with no TMS, and allowed Livingstone to meet the stringent timing requirements of DS1.

The primary lessons of the above discussion are the following:

> *Search and deduction are often essential in reactive systems. Furthermore, search and deduction can be carried out at reactive time scales. Finally, LTMS-style truth maintenance systems can provide an essential tool for speeding up deductive*

search.

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:

- Many reactive system tasks can be carried out using a single model.
- Concurrent transition systems provide an appropriate formalism for building model-based autonomous systems.
- Strikingly simple qualitative models are effective for many real-world tasks.
- Search and deduction are often necessary in a reactive system.
- Search and deduction can be carried out reactively.
- Truth maintenance systems are powerful tools for speeding up search.

6 Lessons from technology insertion

Most of the discussion in this paper has focused on technical issues we encountered while developing the RA. In addition to raising technical issues, the process of working on a real mission and with a real mission schedule provided valuable lessons about inserting this kind of technology into operational missions [2].[5]

Three key technology insertion lessons are the following:

- **Human-centered operations:** While new classes of missions may require systems with highly autonomous capabilities, it is important that such systems also support operational modes in which humans exercise tight control over the system.
- **Validation and Testing:** A major barrier to increasingly autonomous systems is concern about how to test them and validate that they will actually perform as desired. Architectural design choices that let spacecraft engineers focus on the domain model, rather than on the problem-solving methods, can significantly help address this barrier.
- **Schedule impacts:** Putting an autonomous system on-board a spacecraft potentially has a major impact on the traditional flight software development schedule, as it can require knowledge normally codified during operations (after the system is built) to be encoded in the system early on. Developing first things first can alleviate this problem.

[5]Montemerlo [49] provides a set of lessons summarizing earlier experience with technology insertion at NASA.

- **Model-based Skunkworks:** Ensuring coherence of mental models across a large software team can be inordinately time-consuming. This has motivated us to develop a research paradigm in which all software will be programmed in a unified modeling language by a small team supported by automated synthesis techniques and collaborative modeling environments.

We believe that these lessons generalize to other situations in which complex autonomous systems are deployed. We briefly discuss these lessons in the following subsections.

6.1 Human-centered operations

The NewMAAP RA prototype was designed to support scenarios involving extremely autonomous operations in which human communication was impossible. As we moved from prototyping into actual flight code development and teamed with ground operators, we had to extend the RA architecture to address the broader operational context in which the RA would be used. The key insight we gained was that, while extreme autonomy is necessary for certain mission phases, also essential is support for human interaction when such interaction is possible [61, 8]. Such an approach offers two key benefits. First, the ability to draw on human expertise, especially in anomalous conditions, can simplify the design of the system and increase the chance of mission success. Second, designers and operators can automate capabilities incrementally, rather than relying on a fully autonomous system at launch time. This can help increase confidence and improve operator acceptance.

As a result of extensions we made to the RA for these purposes, the RA now has the following features: RA shares the long-range mission profile with ground operators to enable asynchronous ground updates; ground operators can monitor and command the spacecraft at multiple levels of detail simultaneously; and ground operators can provide additional knowledge to the RA, such as parameter updates, model updates, and diagnostic information, without interfering with the activities of the RA or leaving the system in an inconsistent state. Additional forms of support for human interaction with remote agents is a major area of ongoing research [61, 8].

6.2 Validation and Testing

The strict separation between modeling and problem solving heuristics within PS (see Section 3) 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 the RA promise complete autonomy over a much wider variety of complex situations than was previously possible. On the face of it, this makes validation of these systems harder than traditional systems. Fortunately, the declarative nature of AI technology allows the inspection of the models and facilitates a deep understanding of the behavior of the system which is unprecedented in traditional software development approaches.

Our use of a declarative approach dictates a clean separation between models and heuristics. This ensures that system and mission engineers can focus on guaranteeing that requirements are met, and not on the details of how the reasoning engines manipulate the models in order to produce solutions efficiently. A strict separation between models and heuristics allows non-AI specialists to inspect the model and understand the knowledge embedded 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 synthesize problem solving heuristics are important research areas that will widen the applicability of AI techniques to real-world applications.

6.3 Schedule impacts

In the traditional approach to spacecraft development, designers build into the flight software only those capabilities necessary to ensure safety of the system. When anything goes wrong on the spacecraft, it puts itself into a safe state and waits for help from ground operators. This approach enables much knowledge of system interactions to remain uncodified, available only in the heads of the designers. It also enables additional automation to be put into the ground system, typically on a schedule much later than the flight software development.

In the RA approach to building an autonomous system on-board a spacecraft, designers must codify the knowledge at a much earlier stage, so that it can be included in the on-board models used by the RA. This need for earlier modeling can potentially have a major impact on the traditional flight software development schedule. While this is not a technical concern, such schedule issues play a major role in the success of technology insertion, especially in the new era of faster development cycles and concurrent engineering.

Fortunately, we have found that our model-based programming approach has advantages which compensate for the schedule impacts. Since the declarative models mirror the hardware design, the models are easier to maintain in the face of changing hardware details, as compared to traditional software which keeps hardware design assumptions implicit.

With this said, there still remains considerable flexibility about the order in which to perform model development. The key lesson we learned in this respect is work *first things first*: focus first on the critical models at the level necessary to meet launch requirements. Then progressively refine the models to provide increased performance and capabilities. This approach reduces the tendency to have detailed models of some components while major spacecraft capabilities are still unmanaged, and enables the model-based approach to fit into the risk management approach of the overall flight software project.

6.4 Model-based Skunkworks

Often the creativity and speed of a design time decreases exponentially with the size of the team. Lunar Prospector [40] and Mars Pathfinder [14] are two excellent examples of missions developed by small teams, that were inexpensive, were assembled together in a short time span, and operated flawlessly. However it is difficult to sustain this pace as we move towards far more capable missions that come closer to emulating a virtual presence. For example, the DS1 flight software team was comprised of more than 40 individuals, broken into teams responsible for writing hardware specifications, systems engineering specifications, simulators, attitude control codes, discrete device drivers, EXEC procedures, MIR models and PS models, test scripts and systems integration. A significant fraction of the development time was devoted to preparing documents and meeting presentations directed towards knowledge acquisition, scoping, model definition and validation. What made this so challenging is that each of the seven teams had their own mental model of how the spacecraft behaved. The purpose of these time-consuming meetings was to bring these many perspectives into alignment. The large team size and the fact that many of these models were both implicit and changing made miscommunication inevitable, making the task inordinately time-consuming.

The research challenge is then to provide a development paradigm and a set of tools that allow a small team, perhaps a dozen, to develop an equivalent system. These tools should allow models at all levels to be explicit, should facilitate the development of a single coherent model, and should be able to easily track a dynamically changing hardware design. The paradigm we are developing we call *model-based skunkworks.* In this paradigm all aspects of the flight software will be programmed through models. Automated synthesis techniques will use models to generate simulators, discrete flight codes, continuous attitude control codes, and test scripts. To facilitate model synergy, models will be developed using a unified model-based programming language that incorporates the best ideas of encapsulation from classical programming languages. Model building by a disparate team will be facilitated by distributed, collaborative modeling environments. Finally, human assessment of the flight software's capability by systems engineers and project management will be facilitated through analysis tools that generate review documents from models. Finally, an extensive, reusable model library will ultimately allow future spacecraft to be plugged together from past knowledge.

7 Conclusions and future work

The challenge of building a remote agent to assist in establishing a virtual presence in space has proved to be an exciting and unique opportunity for AI. The characteristics of the domain that require highly reliable autonomous operations over long periods of time with tight deadlines, resource constraints, and concurrent activity among tightly coupled subsystems, has led to the development of a Remote Agent architecture based on the principles of model-based programming, on-board deduction and search, and goal-directed, closed loop commanding. The resulting architecture integrates constraint-based temporal planning and scheduling, robust multi-threaded execution, and model-based mode identification and reconfiguration. These components draw upon research in a variety of different areas of AI, including constraint propagation, search, temporal reasoning, planning and scheduling, plan execution, reactive languages, deduction, truth maintenance, qualitative reasoning, and model-based diagnosis.

Jumping headlong to meet fast-paced challenges, first with the NewMAAP demonstration and then with DS1, has provided us with an invaluable opportunity to reassess some of AI's conventional wisdom. Our experience has been in sharp contrast to this conventional wisdom, with the main lessons being that generative planning can scale up to solve practical problems, and that search and deduction can be carried out within the reactive control loop. Furthermore, our embedding in an important real-world problem not only provides a veritable fountain of interesting research problems, but also ensures the relevance of the research. In some sense, this is the most important lesson of our experience!

While the Remote Agent is a significant step toward reaching the goal of providing full autonomy for NASA's explorers, much still remains to be done. Future remote agents will need to be much more *adaptive* in how they react with an uncertain environment. They will need to *anticipate* imminent failures, and plan for contingencies. They will need to be active *information seekers*, to better understand their environment and their own state. As fleets of explorers descend upon distant planets, they will need to *collaborate* with each other to achieve mission goals. We expect future NASA missions, such as the ones highlighted in the Introduction, to provide the concrete challenges that require building more and more capable remote agents. This exciting future is aptly captured by the following vision for autonomy:

> "With autonomy we declare that no sphere is off limits. We will send our spacecraft to search beyond the horizon, accepting that we cannot directly control them, and relying on them to tell the tale."
>
> — Bob Rasmussen, Cassini AACS Cognizant Engineer and New Millennium Autonomy Team.

Acknowledgments

We would like to acknowledge the contributions of the members of the NewMAAP and DS1 Remote Agent teams, without whose tireless efforts this work would not have been possible: Douglas Bernard, Steve Chien, Scott Davies, Greg Dorais, Julia Dunphy, Dan Dvorak, Chuck Fry, Ed Gamble, Erann Gat, Othar Hansson, Jordan Hayes, Bob Kanefsky, Ron Keesing, Sandy Krasner, James Kurien, Jim Larson, Ina Lungu, Bill Millar, Sunil Mohan, Paul Morris, Illah Nourbakhsh, Chris Plaunt, Gregg Rabideau, Kanna Rajan, Nicolas Rouquette, Scott Sawyer, Rob Sherwood, Reid Simmons, Ben Smith, Will Taylor, Hans Thomas, Yu-wen Tung, Todd Turco, Michael Wagner, Greg Whelan, and David Yan. We would also like to acknowledge the invaluable contributions of Guy Man, Gregg Swietek, and Bob Rasmussen for their tireless promotion of spacecraft autonomy. John Bresina, Dan Clancy, Greg Dorais, Ari Jonsson, Jim Kurien, Paul Morris, Kanna Rajan, David Smith, and Rich Washington provided comments on earlier drafts of this paper.

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] Abdullah S. Aljabri, Douglas E. Bernard, Daniel L. Dvorak, Guy K. Man, Barney Pell, and Thomas W. Starbird. Infusion of autonomy technology into space missions – DS1 lessons learned. In *Proceedings of the IEEE Aerospace Conference* [41].

[3] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[4] David Bell, Daniel Bobrow, Brian Falkenhainer, Markus Fromherz, Vijay Saraswat, and Mark Shirley. Rapper: The copier modeling project. In *Working Papers of the Eighth International Workshop on Qualitative Reasoning about Physical Systems*, 1994.

[5] 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* [41].

[6] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[7] Mark Boddy and Thomas L. Dean. Deliberation scheduling for problem-solving in time-constraint environments. *Artificial Intelligence*, 1994.

[8] R. P. Bonasso, D. Kortenkamp, and T. Whitney. Using a robot control architecture to automate space shuttle operations. In *Procs. of AAAI-97*, pages 949–956, Cambridge, Mass., 1997. AAAI, AAAI Press.

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

[10] G.M. Brown, D.E. Bernard, and R.D. Rasmussen. Attitude and articulation control for the Cassini spacecraft: A fault tolerance overview. In *14th AIAA/IEEE Digital Avionics Systems Conference*, Cambridge, MA, November 1995.

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

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

[13] Steve A. Chien and Helen B. Mortensen. Automating image processing for scientific data analysis of a large image database. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8):854–859, 1996.

[14] Richard Cook. Mars Pathfinder mission operations: Faster, better, cheaper on Mars. In *Proceedings of the IEEE Aerospace Conference* [41].

[15] K. Currie and A. Tate. O-plan: the open planning architecture. *Art. Int.*, 52(1):49–86, 1991.

[16] S. Sam Dallas. Space interferometry mission. In *Proceedings of the IEEE Aerospace Conference* [41].

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

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

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

[20] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of AAAI-91*, pages 842–848, 1991. Reprinted in [38].

[21] Johan de Kleer, Alan Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992. Reprinted in [38].

[22] 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.

[23] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. Reprinted in [38].

[24] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, pages 1324–1330, 1989. Reprinted in [38].

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

[26] T.L. Dean and D.V. McDermott. Temporal data base management. *Artificial Intelligence*, 32:1–55, 1987.

[27] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Art. Int.*, 49:61–95, May 1991.

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

[29] 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.

[30] 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.

[31] Tara A. Estlin, Steve A. Chien, and Xuemei Wang. An argument for a hybrid HTN/operator-based approach to planning. In *Procs. of the Fourth European Conference on Planning*, 1997.

[32] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4), 1971.

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

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

[35] 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.

[36] Erann Gat and Barney Pell. Abstract resource management in an unconstrained plan execution system. In *Proceedings of the IEEE Aerospace Conference* [41].

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

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

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

[40] G. Scott Hubbard. Lunar Prospector: Developing a very low cost planetary mission. In *Proceedings of the IEEE Aerospace Conference* [41].

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

[42] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, pages 1194–1201, 1996.

[43] 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.

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

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

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

[47] David S. McKay, Everett K. Gibson Jr., Kathie L. Thomas-Keprta, Hojatollah Vali, Christopher S. Romanek, Simon J. Clemett, Xavier D. F. Chillier, Claude R. Maechling, and Richard N. Zare. Search for past life on Mars: Possible relic biogenic activity in Martian meteorite ALH84001. *Science*, 273:924–930, August 1996.

[48] Andrew H. Mishkin, Jack C. Morrison, Tam T. Nguyen, Henry W. Stone, Brian K. Cooper, and Brian H. Wilcox. Experiences with operations and autonomy of the Mars Pathfinder microrover. In *Proceedings of the IEEE Aerospace Conference* [41].

[49] Melvin Montemerlo. The AI program at NASA: Lessons learned in the first seven years. *AI Magazine*, Winter 1992.

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

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

[52] N. Muscettola, B. Smith, C. Chien, C. Fry, G. Rabideau, K. Rajan, and D. Yan. On-board planning for autonomous spacecraft. In *Proceedings of the Fourth International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS)*, Tokyo, Japan, August 1997. i-SAIRAS.

[53] Nicola Muscettola, Paul Morris, Barney Pell, and Ben Smith. Issues in temporal reasoning for autonomous control systems. In Wooldridge [72].

[54] Nicola Muscettola, Paul Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proc. of Sixth Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, 1998.

[55] David Musliner, Ed Durfee, and Kang Shin. Circa: A cooperative, intelligent, real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

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

[57] 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 Robots*, 5(1), March 1998.

[58] 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.

[59] 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 [72].

[60] 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.

[61] Barney Pell, Scott Sawyer, Nicola Muscettola, Ben Smith, and Douglas E. Bernard. Mission operations with an autonomous agent. In *Proceedings of the IEEE Aerospace Conference* [41].

[62] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987. Reprinted in [38].

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

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

[65] 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 [38].

[66] Ioannis Tsamardinos, Nicola, Muscettola, and Paul Morris. Fast transformation of temporal plans for efficient execution. In *Procs. of AAAI-98*, Cambridge, Mass., 1998. AAAI, AAAI Press.

[67] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 1994.

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

[69] David E. Wilkins. *Practical Planning.* Morgan Kaufman, San Mateo, CA, 1988.

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

[71] 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.

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