Chapter 19 Space Applications

19.1 Introduction

There are several applications of different planning technologies to space exploration. This chapter is mainly devoted to the description of NASA's Deep Space 1 (DS1) mission, a significant application of automated planning to spacecraft. We will described the Autonomous Remote Agent (RA) system, software based on planning and scheduling techniques, which ran on board a spacecraft for several days during the DS1 mission. Our description of DS1 is based largely on the NASA and JPL report [63].

19.2 Deep Space 1

Launched from Cape Canaveral on October 24, 1998, DS1 was the first mission of NASA's new millennium program chartered to test in space new technologies devised by NASA as strategic for future space science programs. The spacecraft was retired in December 18, 2001, after it completed the DS1 mission successfully by encountering Comet Borrelly and returning the best images and other science data ever returned from a comet (Figure 19.1). DS1 successfully tested 12 advanced technologies, including novel electronic, solar, and control devices and novel software components. The RA software system, which comprises automated planning techniques, was successfully tested during an experiment onboard DS1 between May 17 and May 21, 1999.

19.3 The Autonomous Remote Agent

The increased need for a high level of autonomy on board spacecrafts is the main original motivation for the use of automated planning technology in the DS1 mission [63, 410]. In the new millennium, NASA plans to build more and more

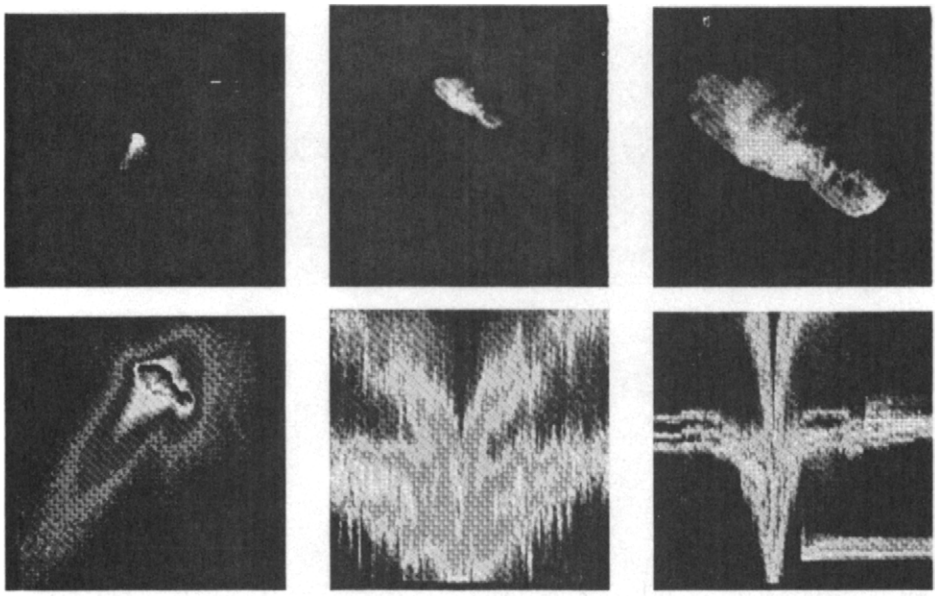

Figure 19.1 Some DS1 pictures of Comet Borrelly.

spacecrafts that must operate without human intervention on board or with limited intervention. Onboard autonomy can reduce the cost of ground control and increase functionality, robustness, flexibility, and adaptation to context.

RA is an onboard software system based on AI techniques. It is able to plan, execute, and monitor spacecraft activities. Its main novel characteristic is that it enables goal-based spacecraft commanding integrated with a robust fault recovery mechanism. A main difference with traditional spacecraft commanding is that ground operators can communicate with RA using goals, such as "during the next period, take pictures of the following asteroids," rather than with detailed sequences of timed commands to the flight software components, such as "at time T1, change the orientation of the spacecraft to D degrees; at time T2, turn on switch number X of camera Y with filter F and brightness level B" and so on.

Given a set of high-level goals, RA is able to generate automatically a plan of actions that achieves those goals and executes the plan by issuing commands to the spacecraft. Actions are represented with high-level tasks that, in order to be executed, need to be decomposed on-the-fly into more detailed tasks and, eventually, into commands to the underlying flight software. The RA component is integrated and can work in closed loop with the underlying real-time execution software that directly controls the spacecraft devices, e.g., the actuators that determine the orientation of the spacecraft and those that control the position of the cameras taking pictures of asteroids. When anomalies are detected (e.g., in the form of failures signaled by spacecraft components or discrepancies between the state expected by RA and the one monitored through spacecraft devices), RA is able to detect, interpret, and respond to failures and anomalies. This can be done either at the execution level or at the planning level, depending on the kind of failure or anomaly. In the case of anomalies that can be repaired at the execution level, RA responds to failures in real time, without any need to suspend the normal activities of the spacecraft. Failures that need to be addressed by replanning require longer response times because RA needs to generate new plans. During the replanning phase, the spacecraft is kept idle in a safe configuration.

RA allows the spacecraft to operate at different levels of increasing autonomy. Operators at the ground station can interact with the spacecraft with detailed sequences of real-time commands. RA executes and monitors the commands through the underlying flight software. This interaction mode allows the operators to interact with the onboard system in the traditional way. Ground operators can also send high-level tasks to the RA on board. RA decomposes the tasks on-the-fly into more detailed tasks and into commands to the underlying flight software. Similarly, ground operators can send goals to the RA on board. RA generates a plan and executes it by monitoring its execution. This is the fully automatic mode.

19.4 The Remote Agent Architecture

The RA architecture is shown in Figure 19.2. RA sends commands to the Real Time Flying Software (FSW), i.e., the software that controls the spacecraft devices (e.g., engines, cameras). It receives data about the actual status of the spacecraft through a set of monitors that filter and discretize sensor values. This communication can be done directly between RA and FSW, or it can be mediated by the Remote Agent Experiment Manager (RAXM), which provides an interface to run-time monitors in the FWS and to the real-time sequencer, which executes detailed sequences of commands. RAXM allows RA to be cleanly bundled on top of the existing flying software and thus makes it reusable in further experiments. Moreover, RA relies on specialized services provided by software modules external to RA, e.g., automated navigation facilities that provide information about asteroid positions.

The RA software is structured in three main components: the Planner and Scheduler (PS), the Smart Executive (EXEC), and the Mode Identification and Recovery (MIR) module. Indeed, RA's ability to provide goal-based spacecraft commanding with robust fault recovery is due to the tight integration of three AI technologies: automated planning and scheduling performed by the PS component, robust multithread execution provided by EXEC, and the model-based fault diagnosis and recovery mechanism of the MIR module. The Mission Manager (MM) is an additional component that stores the mission profile, which contains a set of goals received from the ground station (through the FWS) and selects and

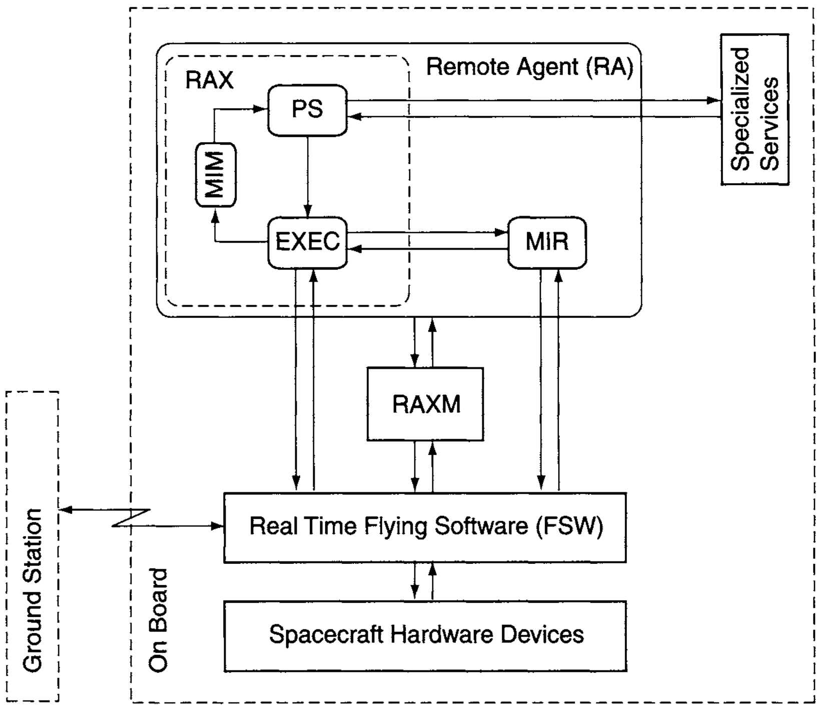

Figure 19.2 RA architecture.

sends to PS the goals related to the period when they need to be processed by PS component. The goals in the MM mission profile can be changed, e.g., on a week-by-week basis. The collection of the PS, EXEC, and MM modules is called the RAX module.

RA translates high-level goals into a stream of commands for FSW in two steps. First, MM selects goals for the next commanding horizon (e.g., two days) and sends them to PS. PS plans for and schedules the tasks that achieve the goals by using its model of the spacecraft. This activity may require that PS gets information from the specialized services. For example, a goal can be described as "perform orbit determination activities for 1 hour every day." PS determines that pictures of beacon asteroids need to be taken. In order to select these asteroids, PS interrogates the specialized service for autonomous navigation. Therefore, the task translates into taking a series of images of the asteroids with a particular camera. Moreover, PS knows that while images of an asteroid are being recorded, the orientation (called more specifically the *attitude*) of the spacecraft must be compatible with the target of the cameras pointing at it. If this is not the case, PS schedules an appropriate turn, changing the orientation so that it is compatible with the cameras pointing at the asteroid.

EXEC is responsible for sending the appropriate commands to the various flight systems it is managing. It is a multithreaded, reactive control system that is capable of asynchronously executing commands in parallel. It does much more than simply executing low-level commands. It is responsible for the following tasks.

- *Executing plans from PS by expanding tasks.* Because PS generates high-level tasks that are not directly executable by FWS, EXEC needs to expand these task at run-time. It provides a procedural language, called ESL [210], in which spacecraft software developers define how complex tasks are broken up into simple ones.
- *Controlling failure handling and recoveries at different levels.* In the event of a failure, EXEC attempts a recovery, either by executing a prespecified recovery sequence, by requesting a failure recovery to MIR, or by requesting a replanning activity to PS.
- *Reactively controlling the execution of commands.* EXEC exhibits its reactive behavior by performing event-driven and conditional commands that depend on conditions that occur at execution time.
- *Achieving and maintaining safe modes as necessary.* As a consequence of its reactive capability, EXEC is able to reach and maintain desired system states by monitoring the success of commands it issues and reactively reachieving states that are lost.
- *Managing time and resources flexibly.* EXEC is able to execute plans with soft time constraints [403], where time deadlines are specified as ranges rather than fixed time points. This decreases significantly the probability of execution failures due to time constraints. Moreover, EXEC manages resources whose constraints have not been resolved by the planner or by the ground operator. This is done by monitoring resource availability and usage, allocating resources to tasks when available, and suspending or aborting tasks if resources become unavailable due to failures [210, 211].

MIR is the component devoted to monitoring faults and diagnosing recovering actions. As each command is executed, MIR receives observations from spacecraft sensors, abstracted by monitors in lower-level device managers like those for cameras, bus controllers, and so on. MIR uses an inference engine called Livingstone to combine these commands and observations with declarative models of the spacecraft's components. Livingstone is composed of two main components: the Mode Identification (MI) and the Mode Recovery (MR) components (see Figure 19.3).

MI is responsible for identifying the current operating or failure mode of each component in the spacecraft, allowing EXEC to reason about the state of the spacecraft in terms of components' modes, rather than in terms of low-level sensor values. MI observes EXEC issuing commands, receives sensor observations from monitors, and uses model-based inference [553] to deduce the state of the spacecraft and to provide feedback to EXEC.

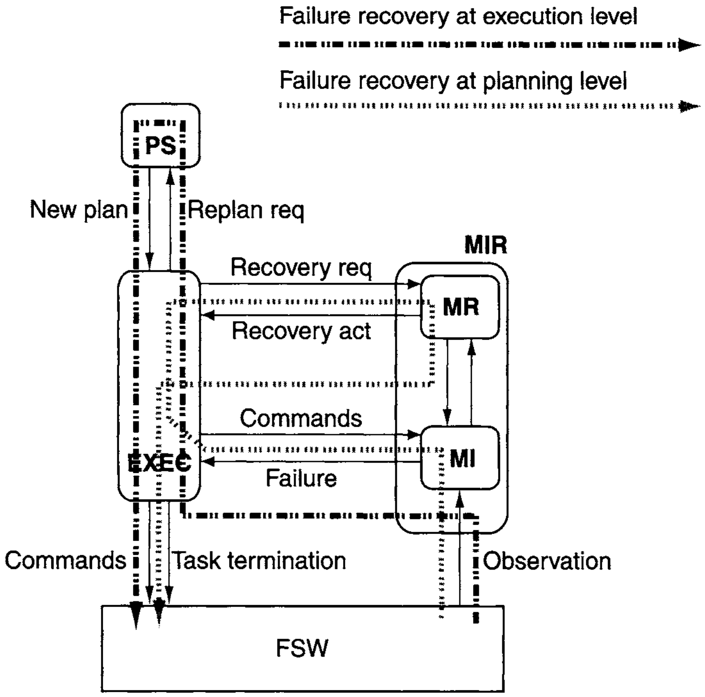

Figure 19.3 Failure recovery at the executing and planning levels.

MR is responsible for suggesting reconfiguration actions that move the spacecraft to a configuration that achieves all current goals as required by PS and EXEC. MR serves as a recovery expert, taking as input a set of EXEC constraints to be established or maintained, and recommends recovery actions to EXEC that will achieve those constraints.

MIR, EXEC, and possibly PS are involved in failure recovery, which can be performed either at the execution level or at the planning level (as depicted in Figure 19.3). The following subsections briefly discuss these two situations.

Failure Recovery at the Execution Level. This situation involves EXEC and MIR in the context of executing a single task generated by PS. Suppose that EXEC is commanding that a camera turn on in order to ensure that the camera is active during a task. EXEC does so by sending an appropriate command to the power driver. MI observes the command and, on the basis of its previous state estimate and its models, predicts the likely next state in which the system will be. Each component state or mode is captured using abstract, or qualitative, models [553], i.e., models that describe the spacecraft's structure or behavior without numerical details. These models cannot quantify how the spacecraft would perform with a failed component, but they can be used to infer which component failed. Livingstone uses algorithms adapted from model-based diagnosis [431, 547] to predict the likely next state and thus the possible failure. In model-based diagnosis, a combination of component modes is a description of the current state of the spacecraft only if the set of models associated with these modes is consistent with the observed sensor values. MI uses a conflict-directed, best-first search to find the most likely combination of component modes consistent with the observations.

The prediction of the likely next state has to be confirmed by MR reading the sensors of the spacecraft, e.g., the switch sensor and the current sensor should be consistent with the camera's power status. If the expected observations are not received, MI uses its model to hypothesize the most likely cause of the unexpected observations in terms of failures of the spacecraft's components. The information about the new state of the spacecraft hardware is sent to EXEC, which now asks MIR for an action to correct the problem. MIR activates MR, which, using the same model, determines the least-cost system state that satisfies EXEC's request and one that is reachable from the fault mode. MIR then gives to EXEC the first action in a possible sequence that will take the system to that state. Such a recovery may involve resetting a device, attempting a command again, or performing a complex reconfiguration of the spacecraft to enable a redundant system. EXEC executes the recovery action (always monitored by MIR) and receives further actions from MIR if needed. When the recovery is complete, EXEC continues to execute the recovery task in a nominal fashion.

Failure Recovery at the Planning Level. This is essentially a replanning activity triggered by MIR and then performed by EXEC and PS. This failure recovery mechanism is activated in the case the failure recovery at the execution level fails, i.e., all the recovery action sequences suggested by MIR fail. For example, if MIR does not manage to recover from the failure to turn on the camera, then it infers that the camera cannot be used. EXEC then knows there is no way to execute the command with success. Therefore EXEC terminates execution with a failure, discards the rest of the plan, and commands the spacecraft to enter an RA stand-by mode. EXEC then asks PS to replan (i.e., to generate a new plan) by passing PS the new initial state. PS receives the goals from MM, generates the new plan, and sends it to EXEC. EXEC exits the stand-by mode, resumes normal operations, and starts the execution of the new plan.

19.5 The Planner Architecture

The architecture of the PS module is shown in Figure 19.4. PS consists of a Search Engine (SE) operating over a constraint-based Temporal Database (TDB). SE begins with an incomplete plan and expands it into a complete plan by posting additional constraints in TDB. These constraints originate from (a) the goals that PS receives in input from MM, and (b) from the constraints stored in a Domain Model (DM) of the spacecraft. DM describes a set of actions, how goals decompose into actions,

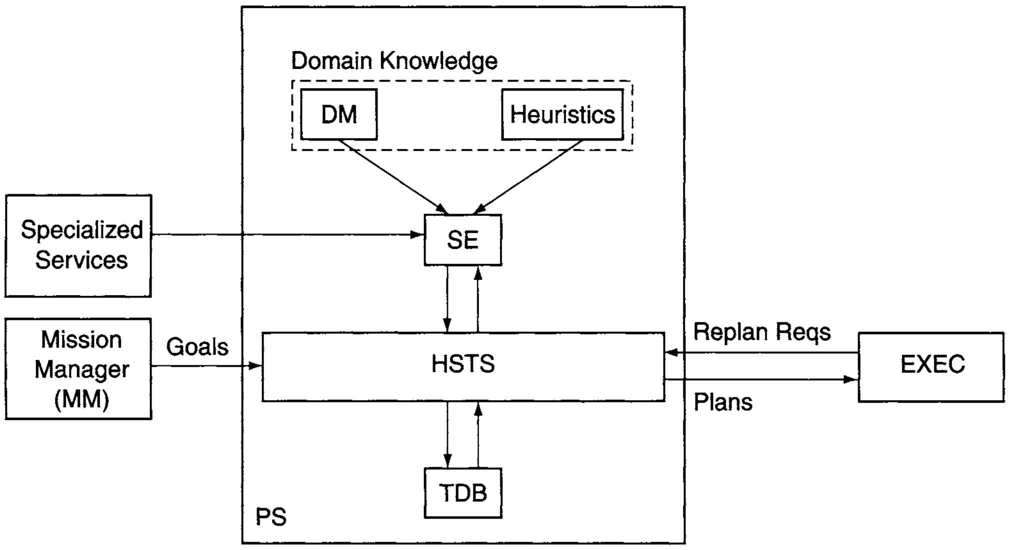

Figure 19.4 PS architecture.

the constraints among the actions, and resource utilization by the actions. For instance, DM encodes constraints such as "do not take images with the camera while thrusting." Access to TDB and DM is provided through the Heuristic Scheduling Testbed System (HSTS).

SE is domain-independent and implements a rather simple search mechanism with chronological backtracking. Because the search space is usually huge and the time allocated to the plan generation activity is limited, a domain-independent search would rarely generate a plan in the required time. Thus a very important component of PS is the module containing domain-dependent Heuristics, which guides the search depending on the domain.

PS tightly integrates planning and scheduling capabilities (see the discussion on this topic in Chapter 14). The planner needs to recursively select and schedule appropriate activities to achieve mission goals. It also needs to synchronize activities and allocate global resources over time (e.g., power and data storage capacity). Subgoals can be generated due to the limited availability of a resource. For example, it may be preferable to keep scientific instruments on as long as possible to gather as much information as possible. However, limited resource availability may schedule a temporary shutdown to allocate resources to mission-critical subsystems. In this case, the result of scheduling generates a subgoal, i.e., "turn scientific instruments off," which requires a planning activity. Therefore PS considers the consequences of action planning and resource scheduling simultaneously (see the general motivations for this choice in Chapter 14).

In the following subsections, we describe in more detail (with some examples) domain models, goals, constraints, plans, and the search algorithm used in the SE module. PS needs to take into account time and resources, where actions can occur concurrently and can have different durations, and goals can include time and different kinds of constraints, such as maintenance constraints. For instance, some control procedures of the spacecraft need to be executed either before, after, or in parallel with other procedures.

- The spacecraft orientation must be maintained so that it points at a direction d for all the time during which the engine is thrusting in a given direction d.
- A procedure that turns the orientation of the spacecraft from direction d_1 to direction d_2 can occur only after a procedure that has maintained the orientation at direction d_1, and it should precede a procedure that maintains the direction at d_2.

Examples of goals are the requirement to achieve the correct thrust vector of the spacecraft engine within a given deadline and the request to take a specified sequence of pictures in parallel with providing a certain level of thrust.

The Domain Model. DM is based on *state variables* and *timelines* (same as in Chapter 14), i.e., histories of states for a state variable over a period of time. Examples of state variables are *engine* and *orientation*. The former represents the status of the engine of the spacecraft, which can be either idle or thrusting in a direction d, i.e., thrust(d). The latter represents the orientation of the spacecraft, which can be turning from a direction d_1 to a direction d_2 (in which case its value is turn(d_1,d_2)) or can be pointing to a fixed direction d (with value point(d)). A further example is the variable *camera*, representing a camera that can take pictures of the surrounding space from the spacecraft. One of its values can be take-picture(b,f), representing the fact that the camera is taking a picture, where the parameter b is the brightness level of a target object and f is the camera filter. Notice that the values of state variables, such as thrust(d), turn(d_1,d_2), and take-picture(b,f), denote the control procedures of the spacecraft, and they contain parameters, e.g., the parameters d_1 and d_2 of the procedure turn(d_1,d_2).

In the following discussion, we call the value of a state variable over an interval a *token*. State variables are piecewise constant functions of time; each such piece is a token. Formally, a token is the tuple (x, p, t_0, t_1), where x is a state variable, and p is a procedure for the state variable x. The procedure p is active in the interval from the start time t_0 to the end time t_1. Thus, for instance, (*orientation*, turn(36, 38), 1,10) means that the spacecraft is changing orientation from 36 to 38 in the period from time 1 to time 10. Tokens can contain parameters, as in the case (*orientation*, turn(d_1, d_2), t_0, t_1), where d_1, d_2, t_0, and t_1 are variables. More generally, a token is $(x, p(y_1, \ldots, y_n), t_0, t_1)$, where $y_1, \ldots, y_n$ are the parameters that denote the variables of the procedure p.

```
(define Compatibility
    ;; compats on SEP-Thrusting
    (SEP-Thrusting ?heading ?level ?duration)
    :compatibility-spec
    (AND
        (equal (DELTA MULTIPLE (Power) (+2416 Used)))
        (contained-by (Constant-Pointing ?Heading))
        (met-by (SEP-Stand-by))
        (meets (SEP-Stand-by)))
)
```

Figure 19.5 Temporal constraints in DDL.

Goals. Goals are represented as constraints on tokens. Given two tokens $T_1 = (x_1, p_1(y_1, \ldots, y_n), s_1, e_1)$ and $T_2 = (x_2, p_2(z_1, \ldots, z_m), s_2, e_2)$, constraints can be of the following kinds.

- *Equality constraints* exist between parameters of procedures, e.g., $y_j = z_k$, with $1 \leq j \leq n$ and $1 \leq k \leq m$.
- *Temporal constraints* exist between the start and end time variables, e.g., s_1, e_1 of token T_1 and s_2, e_2 of token T_2. We allow for the temporal relations of interval algebra, described in Section 13.3.2, e.g., *before*, *after*, *during*, *meet*, and *is-met-by*.

For instance, the constraint that the thrusting procedure can be executed only while the orientation is maintained can be expressed by the constraint *during*$([s_1, e_1],[e_2, s_2])$, where $T_1 = (\textit{engine}, \text{thrust}(d), s_1, e_1)$ and $T_2 = (\textit{orientation}, \text{point}(d), s_2, e_2)$. More generally, goals are conjunctions and disjunctions of these kinds of constraints. Each atomic constraint of a goal is called a *subgoal*.

In DS1, constraints are called *compatibilities* and are encoded in a declarative form called Domain Description Language (DDL). Figure 19.5 shows an example of a DDL description of a temporal constraint. The master token (at the head of the compatibility) is SEP-Thrusting (when the Solar Electric Propulsion, or SEP, engine is producing thrust), which must be immediately preceded and followed by an SEP-Stand-by token (when the SEP engine is in a stand-by mode but has not been completely shut off). The master token must be temporally contained by a Constant-Pointing token. The complete thrusting activity requires 2,416 watts of power. The Constant-Pointing token implies that the spacecraft is in a steady state aiming its camera toward a fixed target in space.

Plans. In DS1, plans are special cases of chronicles, as described in Chapter 14. Candidate plans are refined incrementally by the planner until they are a solution for the given goal. A candidate plan contains the following elements.

- A horizon (h_s, h_e) represents the time window considered by the plan.
- A set of timelines, i.e., sequences of tokens $(T_1, \ldots, T_k)$, contains one token for each state variable x, with $T_i = (x, p(y_1, \ldots, y_m), s_i, e_i)$.
- Ordering constraints enforce that each token in the sequence is followed by the next token, i.e., $h_s \leq s_1 \leq e_1 \leq s_2 \leq \ldots \leq e_{n-1} \leq s_n \leq e_n \leq h_e$.
- A set of equality and temporal constraints enforces required conditions.

The Search Engine. SE starts from an initial candidate plan, which includes a finite horizon of interest (i.e., the time window we want to plan for), an initial timeline (i.e., an initialization token for each state variable), and a set of ordering, equality, and temporal constraints. This specifies the planning problem. It is a special case of temporal planning problems, as specified in Section 14.3.3. It specifies the initial condition with the initial timeline and the goal with the set of constraints. In general, a candidate plan contains *flaws*, i.e., elements that prevent the plan from being a solution plan. Flaws are uninstantiated variables, floating tokens (i.e., timelines with gaps between tokens), disjunctions of constraints, and unsatisfied constraints. The job of plan generation is to resolve the flaws and transform the initial candidate plan into a solution plan. Moreover, the constraints in a candidate plan give rise to a constraint network consisting of the variables in the tokens of the plan and the constraints that link token variables. Constraint propagation techniques (see Chapter 8) are therefore used to eliminate inconsistent values.

Different methods for controlling the search, selecting flaws to be resolved, and propagating constraints can be chosen. In its basic framework, the planning procedure is a recursive function that nondeterministically selects a resolution for a flaw in the current candidate plan. In practice, however, the version of the planner installed in DS1 makes use of a user-defined and domain-dependent search control mechanism that restricts significantly the amount of search needed. A first mechanism to control the search is a *flaw agenda*, which specifies a priority among the flaws to be resolved. The order of the flaws in the agenda defines a priority and a sorting strategy for the nondeterministic choices of the basic plan generation algorithm. A second mechanism is the introduction of new pruning mechanisms in the Heuristics module through a language for the specification of domain-dependent control strategies (see Part III). These pruning mechanisms are at the price of completeness, but they actually make it possible to generate plans that control the behaviors of a spacecraft.

19.6 The Deep Space 1 Experiment

The DS1 mission was designed to validate nonstandard technologies for space exploration, among which were the technologies applied in the RA component. In this section, we discuss the validation objectives, the scenarios, and the experiment results, as well as some lessons learned for the future.

19.6.1 Validation Objectives

The main objective was to demonstrate that RA could autonomously operate the DS1 spacecraft with communication from the ground limited to a few high-level goals for an extended period of time. This translated into specific objectives for RA's main components, i.e., PS, EXEC, and MIR.

- PS validation objectives:
  - Generate plans on board the spacecraft.
  - Reject low-priority unachievable goals.
  - Replan following a simulated failure.
  - Enable modification of mission goals from the ground.
- EXEC validation objectives:
  - Provide a low-level commanding interface.
  - Initiate onboard planning.
  - Execute plans generated on board and from the ground.
  - Recognize and respond to plan failures.
  - Maintain required properties in the face of failures.
- MIR validation objectives:
  - Confirm EXEC command execution.
  - Demonstrate failure detection, isolation, and recovery.
  - Demonstrate the ability to update MIR state via ground commands.

Beyond these main objectives, other validation objectives addressed the impact of the introduction of RA into "traditional" spacecraft software architecture. RA was designed to be integrated with the existing flying software (FSW), and a validation objective was to demonstrate that RA could safely cooperate with FSW and provide a flexibly definable level of autonomy. Even within the scope of the demonstration of the autonomy capabilities of RA, an important validation objective was to demonstrate that adopting RA was not an "all or nothing" proposition and could be commanded by the operator at different autonomous operation levels. Figure 19.6 shows different autonomy levels that were required, from having EXEC issuing low-level commands from a low-level script of commands (level 2, a very "low" level of autonomy), to preparing a plan on the ground station and uplinking

Level	Ground station	Onboard PS	Onboard EXEC
1	Real-time commands	None	None (FSW executes)
2	Command sequence	None	Execute sequence
3	Plan, upload to EXEC	None	Execute plan
4	Plan, upload to PS as goal	Confirm plan	Execute plan
5	Plan with unexpanded goals	Complete plan	Execute plan
6	Goals	Generate plan	Execute plan

Figure 19.6 Autonomy levels of RA. Reprinted from Bernard *et al.* [63] with permission.

it to the spacecraft for execution (autonomy level 3), to providing closed-loop planning and execution on the spacecraft (autonomy level 6). The experiment itself was designed to start working at level 3 and then to smoothly build confidence by migrating to level 6.

Finally, a further objective was to validate the feasibility of the development process of RA from the point of view of development costs, safety, integration with different technologies, reusability of the developed software for different missions, and achievement of high-level confidence by the DS1 team in the technology. Indeed, the RA technology is intended as a tool to support system engineering and operations for a mission—one that can be applied for future and different missions, at a reasonable cost, and with significant advantages—rather than a system that provides autonomous capabilities in a one-shot experiment.

19.6.2 Scenarios

The experiment is organized in three main scenarios: a 12-hour scenario that requires low-level autonomy capabilities, a 6-day scenario that requires high-level autonomy capabilities, and a 2-day scenario that represents a variation of the 6-day scenario.

Scenario 1: 12 Hours of Low-Level Autonomy. In the 12 hours scenario, there is no onboard planning by PS. The plan is generated at the ground station, uplinked to the spacecraft, and executed by EXEC and MIR. The scenario includes taking images of asteroids to support optical navigation and a simulated sensor failure. The planning of optical navigation imaging provides the planner the opportunity to reject low-priority, unachievable goals because, for instance, the optical navigation window had time only to take images of some asteroids.

Scenario 2: 6 Days of High-Level Autonomy. The 6-day scenario includes onboard planning by PS. We describe two parts of the scenario.

At the first horizon, PS generates a plan that includes camera imaging for optical navigation and engine thrusting. Execution of the first plan also includes a ground command that modifies the goals for the second horizon. At the end of the first horizon, PS plans to switch off the cameras. A simulated failure prevents RA from turning off the cameras. This failure is not recoverable at the execution level, i.e., by EXEC and MIR alone, and thus replanning is required. This *failure simulation* demonstrates how EXEC and MIR can make repeated attempts to recover a camera switch until it is deemed permanently stuck. A second plan is generated, and while the plan is executed, the *failure simulation* is undone: the ground station informs MIR that the failure is now fixed, and so the plan execution can succeed.

In the second horizon, another plan is generated and executed that includes engine thrusting, camera imaging, and two further simulated failures, one on a communication bus and another on the command that closes a valve involved in the thrusting process. These failures can be handled by the MIR and EXEC components at the execution level, thus demonstrating the failure recovery mechanisms at that level.

Scenario 3: 2 Days of High-Level Autonomy. This is the most important scenario because it relates to the encounter of DS1 with the comet. This scenario was designed during the experiment in March 1999 to ensure that DS1 could be on track for its Comet Borrelly asteroid encounter in July 1999. It is similar to scenario 2, except for the fact that RA should not switch the camera off after each use due to concerns about thermal effects. In addition, RA was required to produce at most 12 hours of thrusting of the spacecraft in order to assure that DS1 would be on track. A further difference is that the simulated camera switch-off failure was active for the duration of the scenario.

19.6.3 Experiment Results

The flight experiment was run safely on both scenarios 1 and 2. During the most significant scenario, scenario 3, three unexpected circumstances caused some unexpected behaviors by the RA system, which turned out to be the way RA managed to achieve even unexpected validation objectives.

A First Alarm. During scenario 3, PS started generating the first plan. It was generated correctly, but an unexpected circumstance occurred. The PS team monitored the behavior of SE by using a graphical tool, the PSGraph tool. This simple tool shows the search depth versus the number of search nodes visited for each successive step of the planner search. The purpose of the tool is to provide a quick summary of the PS planning process. For example, a trajectory that visits the same depth level several times while the search node number increases indicates that the planner is backtracking. The persistence of this situation for a large number of steps is an indication that PS may be thrashing and that it will be unlikely to return a solution within the allotted amount of time.

The PSGraph plot for the start of the scenario 3 experiment showed a discrepancy between the experiment run a few days before (with PS running on the ground under conditions identical to those of the flight test) and what happened during the flight. This deviation from the behavior of the experiment run on ground indicates that PS on board was backtracking much more! After the deviation, however, the behavior of the planner during the flight started going back to the behavior of the previous experiment. This assured that PS was finding the solution. However, the cause of this different behavior was unknown at the time of the experiment. Later it was discovered that the autonomous navigation system contained information different than expected, therefore generating a slightly different goal for PS. As a result, PS was solving a slightly different problem than the one it had solved on the ground. This unexpected circumstance demonstrated that PS was robust to last-minute changes to the planning goals.

A Second Alarm. The experiment went on smoothly, with RA dealing with the camera switch failures, the resulting replanning, long turns to point the camera at target asteroids, optical navigation imaging, and the start of the engine thrusting. However, at a certain point, it became apparent that RA did not commanded termination of engine thrusting as expected. The experiment was stopped (at this time, RA validation had achieved 70% of the objectives). The problem was then identified as a missing critical section in the plan execution code. This created a race condition between two EXEC threads. If the wrong thread won this race, a deadlock condition would occur in which each thread was waiting for an event from the other. This occurred in flight, even though it had not occurred in thousands of previous trials on the various ground platforms. The DS1 technical report stated that "the occurrence of this problem at the worst possible time provides strong impetus for research in formal verification of flight-critical systems." Indeed, formal methods like model checking have been shown to be effective techniques to discover subtle bugs, e.g., in the form of deadlocks, in concurrent processes.

A 6-hour scenario was then generated to demonstrate the remaining 30% of the validation objectives. This scenario was designed and implemented on the ground station overnight. Once the problem was identified, a patch was quickly generated for possible uplink, but it was not uploaded for lack of confidence in the patch and because the probability of another occurrence of the deadlock was judged to be very low. While it is not surprising that the patch was not uploaded, it was a real win that the DS1 team agreed to run the new scenario. This showed that the DS1 team had developed confidence in RA and its ability to deal with unexpected circumstances.

A Third Alarm. During the last part of the experiment, an unexpected and not yet identified problem occurring between FSW and RAXM caused a message loss. This caused RA to estimate a wrong status for the spacecraft. Fortunately, this discrepancy caused no effects. RA was able to continue running the experiment and to achieve 100% of the validation objectives. RA's vulnerability to message loss was deemed out of scope of the validation.

19.6.4 Lessons Learned

We briefly summarize some lessons learned from the successful experiments.

- The basic system must be thoroughly validated with a comprehensive test plan as well as with formal methods [63, 483]. It is very important to validate the basic system (SE, EXEC, etc.) prior to model validation.
- There is a need for model validation tools. Here there are two main directions. One is the use of automated test-running capabilities, including automated scenario generation and validation of test results. Preliminary work in the area of formal methods for model validation is a very promising attempt in this direction.
- The validation cost of model changes must be reduced as much as possible.
- Tools are needed that support the knowledge engineering process for the construction of the domain knowledge of PS. The effective and easy encoding of domain knowledge, in the form of both domain models and search heuristics, does seem to be the most critical success factor.
- There is also a need for a simple but expressive language to specify goals, including the use of graphical interfaces.

19.7 Discussion and Historical Remarks

We focused this chapter on a particular application, but many other successful space applications deserve to be discussed.

Some of the earliest planning and scheduling applications included automation for space shuttle refurbishment [141]. More recently, Chien and his co-workers [123, 124] present several applications, such as the SKICAT system for the analysis and cataloging of space images and the MVP system for the automatic software configuration of the image analysis. They have also developed two platforms, ASPEN and CASPER. ASPEN integrates various planning tools, ranging from search engines and temporal and resource reasoning to a domain modeling language and a user interface; it is used for spacecraft operations on the basis of spacecraft models, operability constraints, and flight rules. CASPER focuses on integrating planning with execution and plan repair; it has been applied to the space shuttle payload operations [125, 451]. Recent work in the area of planning for science data analysis [237] focuses on near real-time processing of earth weather and biomass data.

Some of the more recent high-profile ground applications include automated mission planning for the Modified Antarctic Mapping Mission [482] and the MER deployment by NASA [27]. It is also worth mentioning that an autonomous spacecraft flight is undergoing, the flight on EO-1 [126]. It is flying from August 2003 for approximately one year (with possibly an additional year). Finally, planning by model checking has recently been applied to the synthesis and verification of software on ground stations [6, 7].

The DS1 mission was one of the most famous applications of planning techniques. Its impact was significant. It was the first time that an automated planner ran on the flight processor of a spacecraft. The experiment, called the Remote Agent Experiment, achieved all the technology validation objectives. It demonstrated the potentialities of planning techniques in a mission that requires full autonomy on board. Among the main success factors, we recall the use of an expressive framework able to represent concurrent actions, action durations, and time deadlines and the use of domain-dependent heuristics to guide the search for a solution plan. Future research at NASA and JPL aims to extend this framework with a tighter integration between planning and execution, including a framework for continuous planning, model-based diagnosis, and reliable verification and validation techniques.

After the DS1 mission, the NASA and JPL teams that participated in the development of RA are now involved in a set of very ambitious programs, from autonomous rover navigation on Mars to the provision of an integrated planning and execution framework for missions in deep space like those for Jupiter's satellite, Europa.