CLOCKWORK: Requirements Definition and Technology Evaluation for Robust, Compiled Autonomous Spacecraft Executives

Mark S. Boddy Steven A. Harp Kyle S. Nelson
Adventium Labs, Minneapolis

Work performed under NASA Grant NAG-2-1634

FINAL REPORT

January 15, 2004

Contents

Chapter 1

Introduction

This report is the outcome of an investigation, funded by NASA's Intelligent Systems program, into methods and means for using compiled automation to speed the development and deployment of increasingly autonomous spacecraft. Many NASA missions currently being planned or under study will require the deployment of an autonomous or semi-autonomous vehicle, which may be a deep-space probe (e.g., Galileo), a plantary rover (e.g., MER), or a lander (e.g., SAGE). Other missions may be possible without a greatly increased level of autonomy, but would still benefit from reduced operating costs, increased safety, and possibly greater science return, should that increased level of autonomy be practicable.

The constraints leading to the need for increased autonomy are two-fold. On the one hand, temporally extended missions employing ever-more-complex vehicles (or worse, multiple vehicles), are rapidly ratcheting up the need for operations support. On the other hand, communications limitations in the form of either bandwidth limitations or light-speed transmission delays mean that effective operation of complex vehicles (a Mars rover, a Europa cryobot, an outer-planets orbiter) require increased on-board decision-making, so that the vehicle is capable of responding effectively to unexpected events without waiting for instructions to be uplinked from mission control.

The kinds of events to which any such system may have to respond include on-board system failures, environmental changes (weather or a landslide), and unexpected results of actions taken by the vehicle, both good and bad. Unfortunately, the techniques most readily adaptable to implementing a high-level "autonomous executive" capable of this kind of reasoning are still close to research, hard to validate in any rigorous sense, and new enough to have little in the way of a track record on previous missions. These issues are cause for serious (and justified) concern among those responsible for mission safety and success. And yet, the missions are still there, still valuable, still needing some form of responsive, reliable, autonomous executive.

The fact that it may be much easier to verify that an artifact has certain desired formal properties than it is to generate an object with those properties has been common knowledge for a long time.[1] This is not to say that verification is always easy, much less that it can easily be done manually. A directly relevant example is Honeywell's Aircraft Information Management System, a real-time, fault-tolerant system certified by the FAA for use in commercial aircraft (e.g., the Boeing 777), in which processor and bus time is allocated according to a schedule that was generated using an off-line, AI-based search algorithm, then verified automatically by an independent tool implemented using more conventional methods, and finally installed on the aircraft as a set of interrupt tables [9].

[1] The entire class of NP-hard problems has this property.

This approach, consisting of off-line generation and verification, followed by installation on-board in a simplified form, has been suggested as a means of accelerating the implementation and acceptance of autonomous systems for NASA missions. Of course, for these domains, a simple table specifying a deterministic schedule will be insufficient. The thesis explored here is that the incremental addition of *compiled automation* to the runtime executives controlling a wide variety of NASA vehicles can be an effective path to providing mission autonomy that is robust, flexible, verifiable and small enough to fit in flight hardware.

We do not intend in this discussion to suggest that the benefits of a compiled approach to automation are not recognized by NASA researchers or mission personnel, or by outside researchers working on NASA-relevant problems. Rather, the intent of this report is to examine a specific range of functions which have historically *not* been compiled, with an eye to how they might be compiled, what the benefits of compiling them will be, and how in specific cases to make tradeoffs as to where to focus effort on providing a function in compiled form.

Specifically, we are concerned with the range of functions falling loosely into areas commonly described as "autonomous executives," "mission planning," or "vehicle health management." These functions are more abstract and more complex than those provided by spacecraft systems concerned with control or sequence execution, but still close enough to the hardware to require guarantees on timeliness and correct response, and in some cases to provide mission-critical capabilities. The need for compilation (or some other way of achieving the same benefits) for these functions is real. The tension between the need for increased autonomy and uneasiness over the difficulty of ensuring predictable behavior from the kinds of algorithms used for higher-level autonomous functions (mission planning, diagnosis, reconfiguration, etc.) has been evident, and increasing, for a decade or more.

1.1 Summary of Findings

The results of our investigation are summarized briefly here. Supporting detail is provided in the rest of the report.

The first point to be made has to do with the level of architectural detail at which it makes sense to talk about compilation. Compilation for autonomy needs to be discussed in terms of individual functions or architectural components, and beyond that, in terms of how those components are implemented (the algorithm used, not just the function performed). Many of the architectures currently being employed or proposed for autonomous spacecraft or rover executives do not appear to break things down to the necessary level of detail to identify individual functions at the appropriate levels. See Chapter 4 for more on this point.

Second, given the current state of the art, compiled automation makes the following properties achievable for a wide range of styles of inference, presented more-or-less in descending order of importance:

Predictability – Inferential models and algorithms can be transformed in ways that permit rigorous guarantees on run-time, responsiveness, coverage (what inputs are accounted for), and correctness (getting the right output for a given input). As discussed in Chapter 5, this can be accomplished for search-based inference and other techniques that have historically not been easy subjects for such guarantees.

Specialization – These functions can be adapted in context-specific ways, for example tailoring a function to a given domain, or compiling a rule-base to optimize a given class of queries. This context arises as a result of the function's interaction with other functions, the architecture

within which it implemented, and the mission it is being implemented to support.

Modularization – A compiled approach supports and encourages the use of architectures in which modules are individually generated and verified, then can be added to a larger system without the need for reverification of the overall system.

Reduced footprint – One motivation for some forms of compilation is a reduction in size, for example to fit a model within strict memory limits. Moore's Law makes this a secondary benefit, but still in some circumstances potentially useful.

Faster inference – Not infrequently, faster models are larger models, because the speedup is achieved through compilation of inference steps into the model itself. Again, while Moore's Law (and the lag in qualifying new hardware for space) means that computing capacities will continue to increase rapidly for the foreseeable future, there are and will remain applications where this issue is relevant (for example, on an atmospheric probe for Venus, or one of the gas giants).

One issue that does not appear much in the current literature on autonomous executives is real-time behavior, by which we mean a rigorous analysis of timing, rather than "it should be fast enough." This analysis can either be done off-line, leading to a derivation of latency and other timing constraints which can then be applied against the implemented system, or on-line, in which timing constraints form part of both meta-reasoning about how long to spend to generate a solution (e.g., a mission plan), and what kinds of timing constraints are required within the plan, given the current context (see, for example [62]).

The questions of what functions to automate and to what degree, and where and how to compile the resulting automation, are complex design questions with implications for the overall mission architecture and design.[2] The tradeoffs involved in making these decisions and currently-available tools to help in evaluating those tradeoffs are discussed in Chapter 6.

Finally, it is our conclusion that substantial operational autonomy for complex spacecraft is well beyond the current state of the art. Hardware failures and other unexpected events can and will lead to significantly altered mission profiles. At the current state of the art, it is difficult to conceive of building an on-board, automated reasoner that could be counted on to respond appropriately to the full range of failures dealt with during the Galileo mission. An incremental approach to increasing autonomy, rather than a "big-bang" move from the current system to full autonomy, is clearly indicated. As an additional architectural implication for spacecraft that are for reasons of distance or other communication barriers unable to phone home, diagnosis and recovery predicated on maintaining some form of homeostasis will in many cases be too limited. This has significant implications for the degree to which mission planning and execution must be integrated, or at the very least coordinated, with diagnosis and recovery.

1.2 Organization of the Report

The rest of the report is organized as follows. Chapter 2 provides a more precise definition of what we mean by compiled automation and presents a brief case study of lessons learned from the Galileo mission, regarding where there were, or would have been, benefits from specific kinds of compiled automation. Chapter 3 discusses overall mission functions, delimiting the scope of this study within those functions. Chapter 4 briefly discusses several architectures or architectural

[2] Some of these implications are discussed in Chapters 7 and 8.

approaches currently being used or under study for use in implementing autonomous executives for NASA missions, with an eye to extracting from those architectures a common set of functions at the appropriate level to consider the benefits (or otherwise) of compilation. Chapter 5 summarizes the current state of the art in methods that can be applied to compiling those functions. We do not provide a fixed mapping of methods to the functions for which they are appropriate, because in the absence of the function's context as described above, there is not much interesting to say about this mapping. Chapter 6 discusses what the tradeoff issues are in deciding, first, whether automation is appropriate and, if it is, whether or not compiled automation makes sense. As a way of providing some concrete information about what the mapping from functions to applicable methods will look like for different mission types, Chapter 7 presents two case studies, including mission descriptions and some specific suggestions for compiled automation within those missions. Finally, Chapter 8 sums up, draws some conclusions, and proposes some areas for further work.

Chapter 2

Compiled Automation

2.1 Definitions and Motivation

Before examining the functions of an autonomous executive and methods for compiling them, we propose some working definitions.

Compilation as an abstract concept denotes the transformation from one form of specification, usually an abstract one, to another form that is suitable for operational use. The dictionary definition[1] gives some sense of the subject, however our intended meaning is broader in its sense of "language", and places a greater emphasis on the distinct requirements of the source (off-line) and operational (on-line) domains. For NASA applications, the on-line portion may be light-hours distant from the compiler. We do not, however, equate the off-line with terrestrial computing, and in some cases it may be sensible for the spacecraft to bring the compiler along.

Source forms for compilation should be those most suitable to human designers, where criteria typically include:

Productivity People want to specify spacecraft behavior in a language that lets them design rapidly and accurately.

Validation One form of representation may be more effective for validation. By validation, we mean "the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model" [2].

Re-use Components or modules in the source form may be composable in ways that are not easy or possible in the target form. This could enable libraries of reusable autonomous function blocks, saving labor on each new mission.

Portability The designer may work at a level of abstraction well above the target platform, providing some insulation from changes to that platform as the mission design evolves.

---

[1] **compiler** ..., 2 : a computer program that translates an entire set of instructions written in a higher-level symbolic language (as Pascal) into machine language before the instructions can be executed. (Merriam-Webster New Collegiate Dictionary.)

Target forms are those that best meet the constraints and goals of the mission and spacecraft. Those include at least:

Reliability The target form must exhibit a reliable and accurate rendition of the source specification, with predictable timing behavior.

Verification One form of representation may be more effective for verification. By verification, we mean "the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model" [2].

Speed There are frequently hard constraints on response latency. Unfortunately spacecraft computing hardware is typically slower than its earthbound counterparts as a side effect of its design for high reliability in the harsh environment of space. Closely related to speed, and uncommon in most terrestrial applications, there may also be a consideration of the electrical energy consumed in the computation.

Storage Current spacecraft hardware typically imposes significant constraints on memory storage. A smaller on-line footprint means more room for primary mission functions, e.g. supporting science goals.

Instruction Set Certain types of compilation may hold special advantages or drawbacks for particular types of target processors, e.g. floating point operations may be quite expensive. On the positive side, it may be possible to compile some functions directly to an ASIC or FPGA.

As we shall see, target forms can be quite varied, including such things as contingent plans, reactive policies, diagnostic rule bases, and Bayesian belief networks. Moreover, the process of compilation may be applied in different ways in different contexts. For example, a rule-base can be compiled in very different ways, depending on whether the limiting constraints are on memory or maximum response time, and perhaps on a desire to optimize response for particular classes of expected queries.

The design problem for compilation for a specific application, then, involves choices for both source and target forms, and the mapping between them.

Autonomy In a broad sense, compilation has been a staple strategy for spacecraft control (e.g. sequence generation) for a long time. In this report, we focus on its application to systems that exhibit a high degree of *autonomy*. Customarily[2] autonomy and the related notion of automation are defined as follows:

autonomy 1 : the quality or state of being self-governing, ...

automation 3 : automatically controlled operation of an apparatus, process, or system by mechanical or electronic devices that take the place of human organs of observation, effort, and decision

automatically (from Greek *automatos*: self-acting) ..., 2 : having a self-acting or self-regulating mechanism, ...

"Autonomy" is a property possessed to varying degree by *agents*, entities with a definable "self". This is conventionally attributed to natural agents such as people, and is conferred, by extension, to robotic agents such as spacecraft, rovers, (or even, feebly, home thermostats). With no desire to

[2]Merriam-Webster New Collegiate Dictionary

explore the philosophical intricacies of agency, we will rely for our operational definition of autonomy on the presence of complete internal circuits that enable unassisted response to changes in the environment (internal or external) of the agent. Artifacts lacking in autonomy must rely on outside agents to wield them. Without its human operator, a remote manipulator arm is little more than a hammer without a carpenter. This report addresses spacecraft that can do quite a bit without explicit external guidance.

Our particular focus is on the implementation of autonomous behavior through compiled means. Many researchers have and continue to extend the boundaries of autonomy in artificial agents, including spacecraft. While we review various actual and proposed architectures for building autonomous systems, it is not our objective to evaluate or critique them. Rather, our intent is to examine how autonomous systems may be more effectively implemented through compilation of various automatic functions within them.[3].

2.2 Galileo Case Study

A review of past interplanetary space missions makes it clear that compiled automation is in common use in the space program already, including functions closely related to autonomous operations, such as sequence generation.[4] This process of compilation is not always explicitely treated as such, nor well-supported with appropriate tools. For example, nearly every software upload to adapt a spacecraft's behavior, whether a response to a problem or a reaction to a newly presented scientific opportunity, can be considered "compiled automation." An off-line system (often the mission team) analyzes the situation, prioritizes potential solutions, and then creates a new set of rules or commands for the spacecraft to execute to adapt to the situation and/or respond to it when the same situation occurs again. Several examples from the Galileo mission are discussed below, showing how off-line analysis led to sequences or configuration information being "compiled" and uploaded to the spacecraft, and under what circumstances this occurred. Unless noted otherwise, these examples are based on the description found in [45].

Galileo was a highly successful mission that explored the Jovian system. It was also a mission that faced numerous technical challenges and overcame many daunting problems. The most widely known of those was the failure of the High Gain Antenna (HGA) to deploy. In an effort to achieve deployment, the mission team radically altered normal cruise operations on Galileo, trying to get the antenna unstuck through such means as shaking (by pulsing the motor), inducing thermal expansion and contraction (alternately heating and cooling the HGA by orienting it towards and then away from the sun), and spinning the spacecraft (all-spun mode) while pulsing the motor. One after another, these troubleshooting steps were developed by the mission team, then reduced to sequences of commands for the spacecraft to execute directly. This is compiled automation in the service of an operational mode involving only limited autonomy—basically, the spacecraft's operations were temporarily disrupted, followed by the resumption of normal operations.

The steps taken to adapt the spacecraft to use the low gain antenna (LGA) as the primary communication mechanism involved more fundamental and lasting changes to spacecraft operations and configuration. The mission team started developing a mission plan with a reduced imaging program (adjusting the mission goals) and, at the same time, investigated means for reprogramming the spacecraft to maximize the information sent via the LGA. The means chosen were image compression and enhancement of the downlink telemetry encoding. These mitigation strategies were reduced (or compiled) into modified software that was then uploaded to the spacecraft, and became part of

[3] *Compiled autonomy* is a term we will reserve for autonomous agents composed entirely of compiled components.
[4] We will make this hazy definition more precise in subsequent chapters.

normal, ongoing operations.

In addition to this reprogramming, several science instruments were adapted by the mission team at different points throughout the mission. For example:

- The dust collector was reprogrammed to take advantage of dust stream data collected by Ulyseses after Galileo was en route to Jupiter (p. 85).
- A radiation-induced fault in the solid-state memory used by the Particles and Fields Instruments caused one 'bit' at a specific memory location to be permanently damaged. The software was rewritten to avoid this address in the future (p. 331).
- A circuit within the gyroscopic system was identified as causing anomolous behavior, possibly because it received more radiation than some others. The software was revised to monitor, detect, and correct the problem (p. 222).

In these cases and others, the mission team conducted an analysis, reduced this analysis to a series of commands that instructed the instruments to behave differently, and uploaded the modified sequences to the spacecraft. In the case of the dust collector, the Ulysses data was not available prior to launch and could not have been forseen. Without the capability to modify the spacecraft's programming remotely, the mission's science results would have been seriously compromised.

Finally, there was a malfunction involving the tape recorder, a piece of equipment that turned out to be a key component of the HGA failure mitigation strategy. Telemetry data suggested that the tape recorder failed to stop after the command to rewind, transitioning from fast forward to fast reverse. Studying the scheduled activities and the associated telemetry, the team decided that while the mechanism was active, the tape failed to move when the command was received. Based on this information and some additional analysis, the team redefined the 'start point' of the tape and incorporated some other changes into the scheduled housekeeping and science activities of the spacecraft. In order to protect from the tape detaching from the reel, it was also decided that the 'built-in' tape recorder functions that were likely to cause that event would instead be executed one step at a time under the control of the spacecraft computer. Consequently, new software was uploaded to monitor the tape's condition and automatically recover when it got stuck.

Perhaps the most extensive instance of compiled automation on Galileo was the process of creating, verifying, and executing the science and housekeeping tasks during each orbital tour. On each pass, Galileo was given a sequence of commands that detailed what it was supposed to do and when it should do it during a given encounter. Ideally, the sequence would be executed autonomously. The Sequence Integration team, consisting of 25 engineers, started with the activities necessary to keep the spacecraft healthy and then worked on the activities necessary to meet the science objectives.

The science objectives were decided within several working groups and then conflicts were iteratively resolved by these teams. Once a proposed sequence was developed, specialists analyzed it in detail to make sure that no action would cause a problem for his or her own particular subsystem. A higher level analysis eliminated conflicts (e.g., two instruments needed the solid state data buffer at the same time, causing an overwrite) and looked for opportunities to maximize science. Each sequence took about 2 months to prepare, matching the period of Galileo's orbit (p. 130).

Part of the development process was a contingency analysis where the team did 'what-if' exercises on the generated sequences. It was because of these exercises that a radiation induced memory failure was quickly and accurately mitigated.

From these examples, we draw the following conclusions. First and foremost, substantial operational autonomy for complex spacecraft is well beyond the current state of the art. Hardware failures

and other unexpected events can and will lead to significantly altered mission profiles. At the current state of the art, it is difficult to conceive of building an on-board, automated reasoner that could be counted on to respond appropriately to the failures dealt with during the Galileo mission, making the full scope of operational and configuration changes required. An incremental approach to increasing autonomy, rather than a "big-bang" move from the current system to full autonomy, is clearly indicated. As an additional architectural implication for spacecraft that are for reasons of distance or other communication barriers unable to phone home, diagnosis and recovery predicated on maintaining some form of homeostasis will in many cases be insufficient. This has significant implications as well for the degree to which mission planning and execution must be integrated, or at the very least coordinated, with diagnosis and recovery.

Second, sequence generation as described above for a single spacecraft of even moderate complexity is currently a very time-intensive job. According to [45], the Galileo mission had at least 25 people working on defining and validating the orbital sequence. If we assume that these 25 people were full time on that job, it means that each orbital sequence took *4 person years per encounter* to generate, validate, and verify.

Once created, this sequence was tested, then uploaded to the spacecraft. This is exactly the process of compilation as we have described it, but using a time-intensive, highly manual process for which there are limited tools to generate and validate the compiled artifact. Automating this process in predictable, verifiable ways would be a huge win, whether or not the end result was to put the planning process on-board.

Finally, the feasibility and utility of the process we are arguing for, of compiling models, rule-bases, algorithms, or sequences to be uploaded to the spacecraft, then validating the compiled objects rather than the algorithms used to generate them, is well-supported by NASA's current processes, in the "what-if" contingency analysis described above.

Chapter 3

Scope of Study

The major functional requirements of spacecraft determine the sorts of functions, or combinations of functions, for which we are interested in finding methods of compiled automation. The functions of interest are indicated by the shaded region in Figure 3.1. This shaded region sits on top of spacecraft systems and functions that are already substantially automated (and in many cases compiled), but is still close enough to the hardware to require guarantees on correctness, coverage, and timely behavior.

3.1 Mission-Level Functions

Several inter-related functions influence operation of the vehicle(s), but are best considered at the mission level. For example, to properly fulfill a science objective, the spacecraft needs to be in the right place, with the right orientation, with appropriate instruments powered, communication channels open, and data processing available.

- **Mission Planning and Scheduling** encompasses developing a plan that satisfies the specific science and other mission goals, reducing it to a partially ordered sequence of actions, and assigning resources and times (or time constraints) for execution (i.e., the schedule).
- **System Health Management** includes diagnosis (or prognosis) of vehicle malfunctions, and determining how to reconfigure the spacecraft to compensate for a failure or a change in the mission goals.
- **Reconfiguration** is the act of modifying a spacecraft to compensate for a failure, take advantage of a new opportunty, or transition to a different mission phase for example from a flyby to an orbiter or an orbiter to a probe.
- **Multi-Vehicle Coordination** refers to coordinating several vehicles that are part of a mission. For example, long baseline interferometry experiments, constellations of vehicles, or coordinated lander/orbiter operations.
- **Validation and Verification** includes validation at design time as well as verification of modifications created during the mission to address problems or capitalize on opportunities.

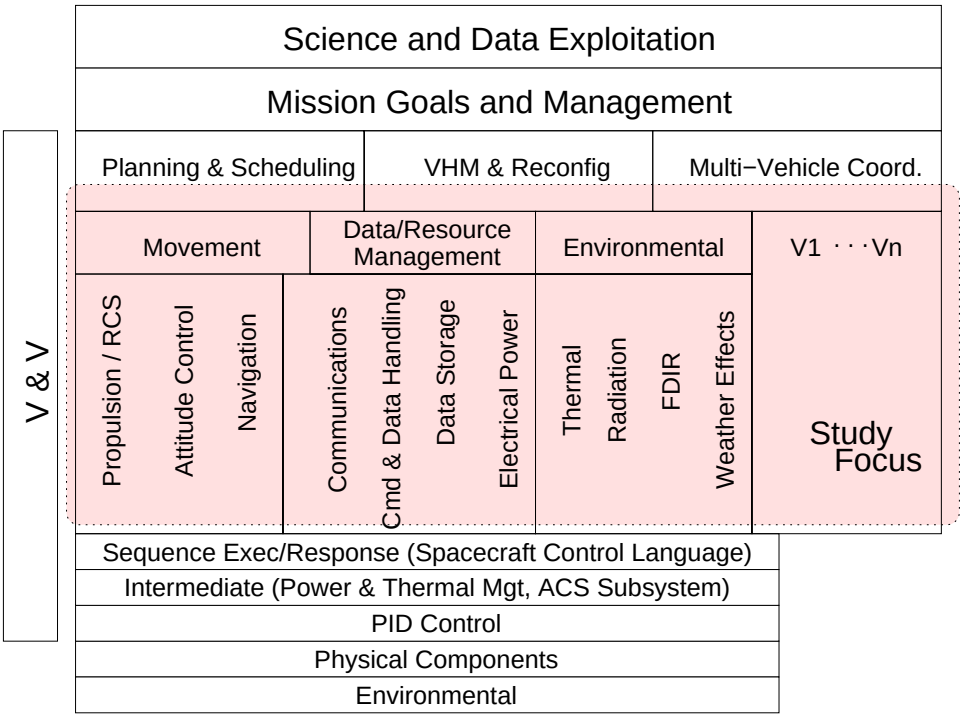

Figure 3.1: Notional spacecraft architecture showing the scope of the study.

These mission-level functions interact in complex ways. For example, outputs of the SHM fuctions directly feed into the creation or selection of contingent plans or schedules. V&V of the spacecraft components is related to confidence in executing a schedule as defined. Multi-vehicle coordination can be used as a tool in mission planning, as well as in recovering from failures.

3.2 Spacecraft-level Functions

Executing the decisions made by the mission-level functions eventually falls to one or more of the spacecraft that comprise the mission. Spacecraft can be partitioned into three general categories based on mission operations: Flyby, Orbiter, Surface [26].

- Flyby spacecraft are those vehicles whose primary science mission involves an approach to an object or set of objects on a continuing journey. Some examples include Voyager, Deep Space 1, and Giotto.
- Orbiters are spacecraft that enter into orbit around an object for an extended period of time. Examples include Clementine, Magellan, and NEAR.
- Surface spacecraft include those vehicles which enter the atmosphere or land on an object for the purpose of conducting science. Examples are: Mars Pathfinder, Pioneer Venus probes, and the Galileo atmospheric probe.

Often a given mission can be described as fitting within different categories, depending on mission phase. For example, while enroute to its final objective of becoming a Jupiter orbiter, the Galileo

spacecraft completed a successful flyby of the planets Venus and Earth and asteroids Ida and Gaspra (it collected scientific data during these encounters) [45]. The NEAR Shoemaker spacecraft both orbited an asteroid and subsequently landed on it, returning scientific data after its successful landing [66]. In the case of NEAR, it was not originally intended to be a Surface spacecraft; this was decided after arrival at asteroid Eros.

Regardless of its mission category, every spacecraft must support functions to navigate, control its attitude, communicate to Earth or another spacecraft, manage its resources, operate its scientific instruments, diagnose and recover from failures, and perform maintenance to maintain its health and maximize its longevity. We categorize these functions as follows:

Movement encompasses those functions required for the spacecraft to track location and velocity (Navigation), have the means to estimate and control its orientation (Attitude Control), and change speed and direction (Propulsion).

- **Navigation** refers to those functions necessary for the spacecraft to navigate in its environment. In the case of Orbiters, the navigation component generally contains a star tracker or other reference unit to estimate position and detailed knowledge of orbital mechanics or, in the case of a surface spacecraft, a detailed knowledge of the topography to figure out where it is and how to get to where it needs to go.
- **Attitude Control** refers to those functions of the spacecraft to attain and maintain a commanded orientation in space. For orbiters and flyby missions, attitude control may be served by inertial reference units or star trackers (orientation), a reaction control system to effect changes in orientation, a momentum control system (e.g., reaction wheels or control moment gyros), so the spacecraft can remain stable. Surface spacecraft might implement flight control surfaces, adjustable appendages, or other mechanisms to adjust its orientation.
- **Propulsion** refers to the functions necessary to change a spacecraft's velocity (magnitude and/or direction), either in space, in an atmosphere, or on the ground. Propulsion can be exhaustible (e.g., solid fuel rockets), renewable (solar powered electric motors), inexhaustable (e.g., the solar wind and or ion propusion), or opportunistic (e.g., gravity assists including around planets or down a terrestrial hill). Clearly, attitude control and navigation are tied quite closely to propulsion in that propulsion is often the means by which the spacecraft can be moved and oriented.

Data and Resource Management refers to those capabilities that must be shared among several components of the spacecraft and that may be consumable or are otherwise limited in availability. These include:

- **Communications** refers to data and information being transmitted to and from the spacecraft. This can be from the spacecraft back to Earth or from a spacecraft to another vehicle (e.g., surface vehicle to an orbiter). Important issues include Deep Space Network (DSN) availability and capabilities, transmission power and bandwidth tradeoffs, redundancy, blackout periods, and round trip light time.
- **Command and Data Processing** and **Data Storage** relate to the processing and storage of science data, telemetry, and commands on, or for, the spacecraft. Issues relating to data storage and processing after it arrives back on Earth (i.e., received by DSN) are also important, but are out of scope for this study. The topics are considered together because there often is a tradeoff between data processing and storage, in that you can use some storage space for processing and vice versa, especially when using FPGAs or other reconfigurable processors.

- **Electrical Power** are those functions related to ensuring adequate electrical power and efficient use of it primarily for the science instruments to function and return the maximum amount of data. Often only a very modest power budget is availabe for these operations and the mission team and spacecraft must be careful about its use as an oversubscription could generate a failure and lead to a "safing" event.

Environmental are those functions necessary for the spacecraft to perform in its environment. Not all elements of this environment can be anticipated prior to launch. A close relationship exists between these functions and the higher level SHM and Reconfiguration functions. The environmental functions include:

- **Thermal and Radiation** are those functions that shed excess heat and generate additional heat when required. Many thermal functions are satisfied with passive approaches, such as heat sinks and thermal blankets, while others are active such as retractable heat shields or heaters. Often, the spacecraft orientation and configuration can be changed to help facilitate thermal and radiation control.
- **Weather or other Atmospheric Effects** are those functions designed to adapt the spacecraft to the effects of weather or the atmosphere. Rovers may contain specialized functions that provide stablity in high winds, measure and adjust for water currents, deploy shields during a sand storm, etc.
- **Fault Detection, Isolation, and Repair (FDIR)** are functions that react to detected failures. It is distinct from SHM in that SHM considers mission goals, while FDIR simply reacts to the detected problem. FDIR is included as part of the environmental functions because FDIR mitigates locally caused failures such as particle impacts, single event upsets, fuel tank leaks, etc.

3.3 Out Of Scope

The items at the top of the figure (Science and Data Exploitation and Mission Goals and Management) are not of present interest as they tend to be exclusively ground operations, generally off-line, and largely manual. While some techniques for compiled automation may be of assistance (e.g., automated data processing), these are not primary drivers. The lower part of Figure 3.1 is likewise not in scope since the use of automation in these functions is relatively mature (see, for example, Spacecraft Control Language [47] and related work on Earth Observer 1 [78]). Classical control algorithms are likewise out of scope.

3.4 Relationship to Compiled Automation

Understanding the spacecraft-level functions and how they vary with respect to spacecraft category is important to understanding the potential benefits of compiled automation, and how those benefits vary depending on mission parameters. For example, in the case of a Surface spacecraft like the Pioneer Venus Multiprobe mission, the Galileo atmospheric probe, or future missions like those that propose to deliver an airplane or glider to a planet, the mission duration (e.g., the descent) and time constraints (e.g., real-time flight) are so short that Round Trip Light Time (RTLT) makes detailed command and control from Earth infeasible. On the other hand, surface vehicles on more benign environments like the Moon or Mars can conceivably be commanded and controlled from Earth,

though the lack of detailed situation awareness and the lightspeed lag, at least for Mars, may slow operations and reduce the efficient collection of scientific data.

As a mission proceeds and the spacecraft moves between mission phases, the requirements and techniques for compiled automation will vary. That is, an approach suitable during a planetary flyby may not be suitable, or desirable, during the Orbiter or Surface phase of the mission. During the orbital insertion phase of Galileo (changing from Flyby to Orbiter), the spacecraft was configured to ignore many of the conditions that would have caused the spacecraft to safe itself during its Flyby phase [45]. This was because missing a critical burn or attitude adjustment would risk the entire mission, whereas during flyby or while in orbit, there was time to recover, if and when the spacecraft safed itself.

The extent to which compiled automation delivers benefit for each of the spacecraft functions will vary with the mission in ways we explore further in Chapter 7. The architecture of the spacecraft itself can have an important effect on the suitablity of compiled automation. Aspects of this are considered in Chapter 4.

Chapter 4

Autonomy Architectures

The mission-level and spacecraft-level functions outlined in the previous chapter are monitored, controlled, and executed by the spacecraft executive. This chapter provides a synopsis of proposals for the organization and implementation spacecraft executive functions.

The architectures included here are generally intended for unmanned spacecraft that will operate with a minimum amount of human involvement, save for humans providing supervisory level directions to the spacecraft from Earth. A number of current and future space missions have the human co-located with the spacecraft, i.e., crewed presence in space. In these cases, the automation architecture selected for the spacecraft will necessarily have to include the role of the human (both the Earth-bound mission team and the spacecraft crew) in conducting the mission. Most of the issues identified in Chapter 3 remain relevant here, with the proviso that appropriate feedback to the crew, flexible mixed-initiative task support, and other human-oriented topics will need to be integrated into the solution. These matters are beyond the scope of the current work.

In most of the architectures discussed in this chapter, a *layered* architecture is visible in some form, with deeper layers handling more elementary tasks. A standard 3-layer architecture has layers for planning, execution and control. However, the lines between the layers can be drawn at varying depths, and in some cases are blurred. Furthermore, depending on the mission and specific implementation, the layers might be implemented on the vehicle, on Earth, or even on an intermediate vehicle in space (for example, an orbital relay for a lander, rover, or probe).

While it is common to define architectures in layers, this is a different decomposition than one might arrive at from a consideration of a functional architecture such as that in Chapter 3. Layered architectures, including those described here, may not do a good job of capturing the details of data or control flows, which may include paths off-board as well as on-board.

Basic to any control loop, from the lowest level to the highest, is some form of *feedback*: future actions need to be conditioned on the results of previous actions. We follow [15] in breaking this feedback loop down as an "OODA" loop (Observe/Orient/Decide/Act), Figure 4.1. Specifically, these phases apply to autonomous executives as follows:

Observe – sensor pointing, local smart processing, possibly some local smart processing or (very) local sensor fusion. For the most part, sensors are maintained and controlled at a level below our scope in this study. There are exceptions, for example the need to use a camera both for science operations and landmark tracking on a rover, or star-tracking using a science instrument due to failure of the primary sensor.

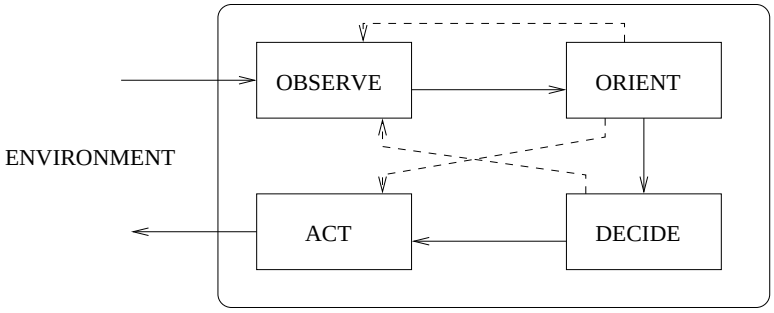

Figure 4.1: The OODA loop (after [15]).

Planning and acting to gain information has an interesting relationship to this loop. One way to think about it is that "decision" and "action" put the robot in a situation and configuration where "observe" magically gets the right information, but then what about repeated sensing operations for triangulation?

Orient – situation awareness, sensor data processing, diagnosis, mode/state identification.

Decide Planning, reactive or projective; policy generation and modification, sequence generation. One interesting issue here is that deciding and acting may be hard to separate temporally, especially as you cross levels in the architecture. For example, in the Remote Agent architecture, the Planner/Scheduler generates plans ("Decide"), which are then passed to the Executive to, well, execute ("Act"), but in the course of doing that the Executive does some rule-based task expansion and resource assignment (which looks like "Decide" again).

Act Execution of a policy, reactive plan (e.g. RAPS, or the Remote Agent executive), or a sequence.

Boyd's intent in framing this loop, and ours in using it, is to capture the structure of higher-level, more abstract and more complex feedback than the simple propagation of a scalar error signal in a numerical controller. Its relevance to the current study is that functions or combinations of functions to be considered for compilation must be defined not only by level in the hierarchy, or by which functions they satisfy, but by what part of this loop they support.

In the rest of this chapter, we briefly discuss different architectures for autonomous executives, drawing some conclusions from their common elements regarding likely functions for compilation.

4.1 Remote Agent

Remote Agent was an ambitious experiment to demonstrate autonomous spacecraft operation. [60, 5, 58] The ultimate goal of this effort was to build a spacecraft that could function autonomously for extended periods, with high reliability, executing concurrent tasks in a tightly coupled system. A version of Remote Agent flew on the Deep Space-1 mission to Comet Borrelly in 1998, and was tested in May of 1999. The Remote Agent architecture is shown in Figure 4.2. It incorporates constraint-based planning and scheduling, a multi-threaded executive, and model-based mode identification and reconfiguration.

DS-1 was a relatively complex spacecraft on a moderately long-duration mission. Consequently, one of the primary functions supported by Remote Agent was diagnosing and coping with spacecraft hardware failures. In the flight tests, simulated faults were injected to determine how Remote

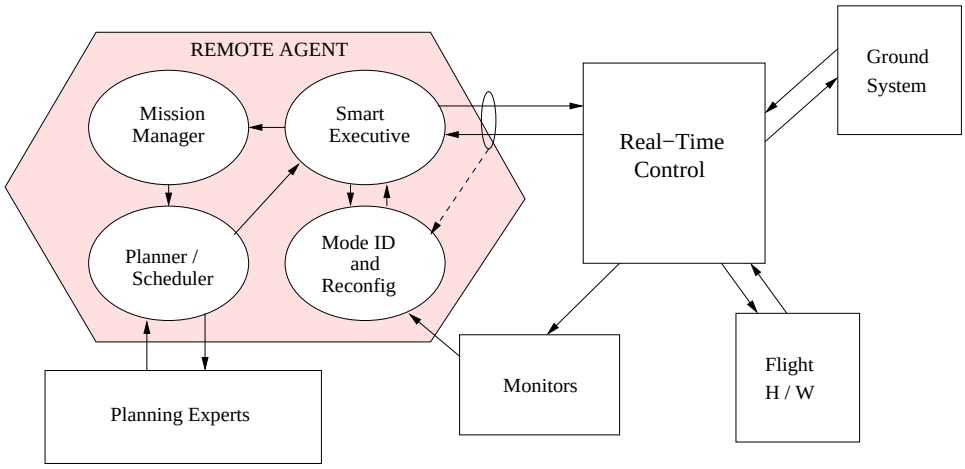

Figure 4.2: The Remote Agent architecture in the context of other spacecraft systems (after Muscetolla et al. 1998).

Agent would resolve or work around them. Facing the question of how much to do on board in real time versus on the ground at compile time, the developers of Remote Agent noted that on a long mission a large number of failures can be expected to appear: "Hence pre-enumerating responses to all possible situations becomes intractable." [60] (p11). The philosophy of Remote Agent leaned heavily on reasoning from internal models. Its creators acknowledge that this runs counter to conventional AI wisdom regarding responsive robot control.

The planning and scheduling component (PS) of Remote Agent used a general purpose planning engine that employs iterative refinement (backtracking) search [49]. PS is a purely generative planner, constructing plans from atomic actions, working at a single level of abstraction. The Domain Description Language (DDL) was used to capture the state variables, constraints, resources and operations. DDL provides a unique homogeneity: both states and actions are consolidated under the notion of "token". Plans are rendered as the evolution of timelines for state variables. In the DS1 Remote Agent experiment, there were 18 state variables and 42 token predicates. The largest plan generated consisted of 154 tokens and 180 temporal constraints. Planning in Remote Agent is done in cycles, each plan specifying the operations for a finite horizon, typically several days. In the case of Deep Space One, Remote Agent takes about 4 hours to produce a 3 day operations plan on a 25 MHz processor.

The mode identification and reconfiguration (MIR) component of Remote Agent was provided by Livingtone, a discrete model-based controller. Livingstone combines aspects of classical feedback control and AI planning to try to put the spacecraft into the minimal cost configuration that meets the desired goal. The same declarative spacecraft model was used by Livingstone for both deduction and reconfiguration. Dynamics were provided by a temporal logic that modeled transitions between modes in various circumstances (nominal, executing command, failure). Livingstone models were compiled into propositional constraint networks to speed best-first search.

The execution system (EXEC) uses a procedural language, ESL [38], to robustly execute the plans from the planner/scheduler. It periodically asks for new tasks as needed. Plans are decomposed by EXEC into commands for the real-time system, thus allowing runtime decisions as to what method or what resources may be used to pursue some high level planned activity. EXEC operates within the degrees of freedom in the plans, such as slack in the allowable execution time.

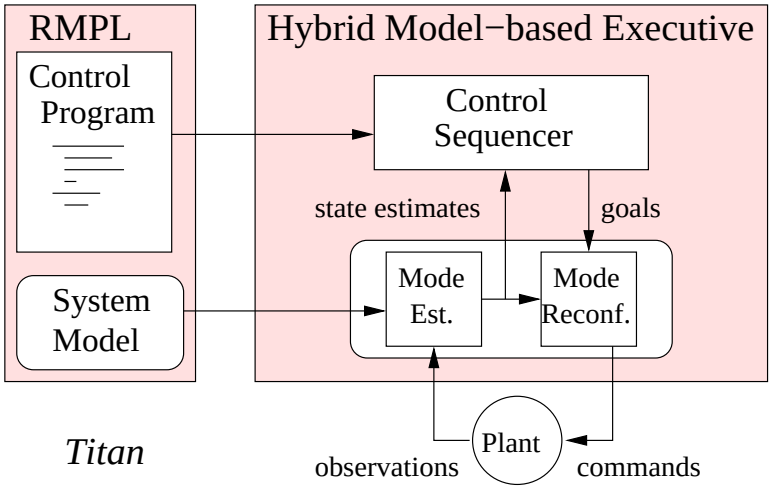

Figure 4.3: The TITAN architecture for model-based programming, (after Williams, 2002).

Remote Agent researchers suggest that "search and deduction are often essential in reactive systems", and that the demands to react in a time scale of tenths of seconds (with a slow cpu) can be met by systems that do online search.

4.2 Titan

The Titan system implements a paradigm for autonomous systems called *model-based programming* [84]. In contrast with traditional embedded programs that force programmers to deal directly with the sensors and actuators of the system, the model-based programming paradigm lets programmers view the underlying system in terms of estimated state and goals. The Reactive Model-based Programmming Language (RMPL) provides the language for expressing activities at this level. When combined with a system (plant) model, the program and model can be compiled to a form that can be executed efficiently by a runtime executive. The architecture for this system consists of a model-based program running on a hybrid model-based executive as shown in Figure 4.3.

The Titan executive consists of a system-level control sequencer and an engineering-level deductive controller. The sequencer runs a compiled RMPL program, in the form of a hierarchical constraint automaton (HCA). Note that the compilation process can occur offline, and the RMPL program need not be embedded in the target spacecraft. The sequencer receives state estimates from and sends state goals to the controller. The controller uses the system model to estimate the current plant state from observations and deliver commands to reconfigure the hardware to meet goals. Titan is a superset of the Livingstone component of the Remote Agent system. The "Burton" reactive planner (RP) for Titan is discussed in [85].

RMPL has been extended in various ways, notably to handle fast temporal planning [51]. In this extension, RMPL programs are compiled into temporal planning networks (TPNs) which are analagous to the plan graphs used in planners such as GraphPlan [8]. In TPNs, linked node pairs represent activities, and choice points by individual nodes. The graph is augmented by temporal constraints (bounding activity duration) and symbolic constraints (expressing state conditions that must hold for a plan to be valid). *Kirk* is an RMPL interpreter that finds an unconditional temporal plan that is both complete and consistent, i.e. all choices have been made at each decision point, and no constraints have been violated. Kirk begins with the compiled TPN representation and outputs a

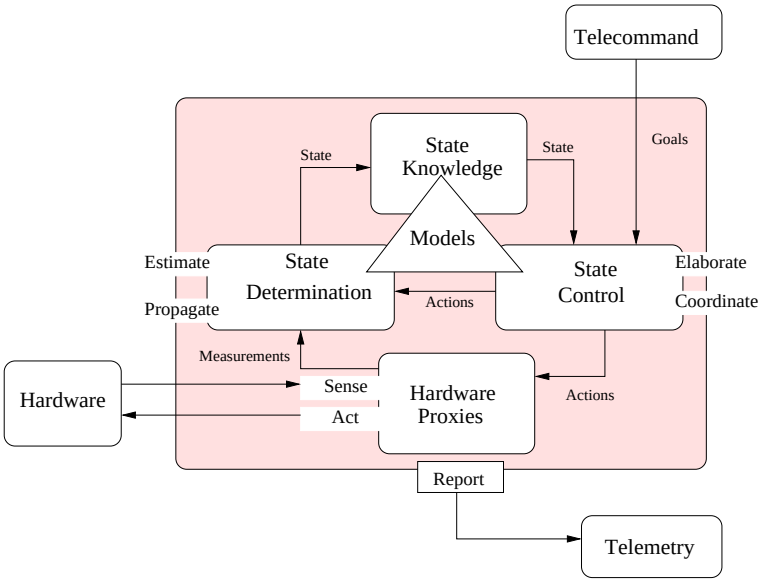

Figure 4.4: The MDS high level architecture, (after Rinker, 2002).

set of acceptable paths through the network.

4.3 MDS

The Mission Data System (MDS) is a software architecture designed to specify an end-to-end information system for ground, flight and test [29, 31, 70].

The high level architecture for MDS is shown in Figure 4.4. MDS encourages the use of integrated planning and scheduling to achieve autonomy. Goals, intents, and constraints are to be made explicit in a declarative manner [74]. The underlying approach is similar to Titan in its emphasis on inferring and controlling system state. State determination is architecturally separated from control, and uncertainty of state estimates is acknowledged.

Control in MDS entails imposing temporal constraints on states. Goals are formally state constraints (limits on the range of a state variable) between two time points. Temporal constraints limit the duration between two event times.

MDS does not stipulate a specific algorithmic solution, but provides a framework consisting of core classes to be adapted by each project that deploys it. The modeling language UML provides a standard for this effort.

4.4 CLARAty

The Coupled Layered Architecture for Robotic Autonomy (CLARAty) is an architecture that has been developed at JPL for robotic control [81, 33, 32]. The top-level CLARAty architecture is shown in Figure 4.5. It is a two layer architecture comprised of a decision layer and functional layer. The functional layer provides a set of standard robot capabilities connected to the robot hardware. The decision layer contains a planner and executive which are tightly coupled and interfaced to a global

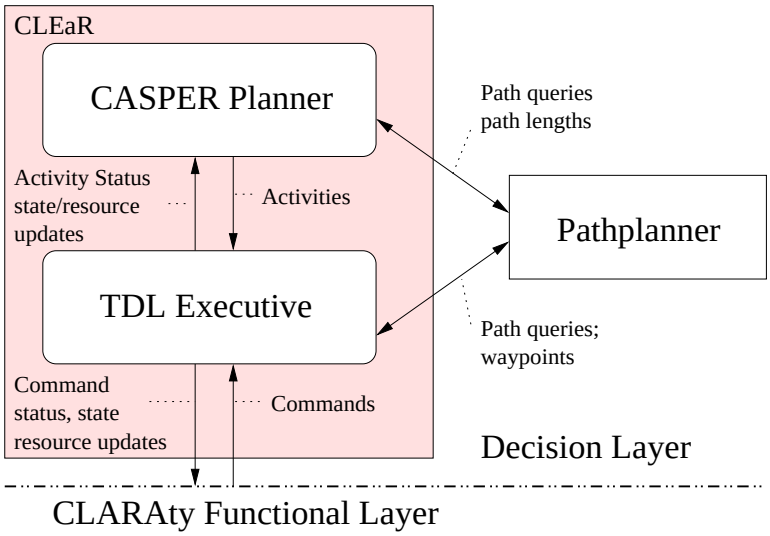

Figure 4.5: The CLARAty planning and execution system architecture (after Estlin et al. 2002).

path planner.

The Continuous Activity Scheduling Planning Execution and Replanning (CASPER) provides the planning function for the decision layer [18]. CASPER reasons about spacecraft operations using models expressed in the ASPEN modeling language (AML). Model elements include activities, parameters, parameter dependencies, temporal constraints, resources, state variables and reservations. Activities are schedulable ocurrences that may be decomposed into hierarchies of subactivities in various ways.

CASPER uses an iterative repair algorithm that fixes conflicts one at a time by making incremental repairs to the existing plan [73]. This continuous approach contrasts with generative planning in that plans can often be updated or repaired in seconds, whereas a full replan on failure may take minutes to hours. Replanning is triggered whenever the current plan is projected to be infeasible given the current state. CASPER relies on most-committed local heuristic search for finding solutions to problems like resource conflicts; some of the heuristics it deploys are domain-independent while others may use domain knowledge.

The executive role in CLARAty is served by the Task Description Language (TDL) system. TDL translates tasks into low level commands and also performs exception handling and task synchronization. The executive is intended to provide regular feedback to the planner about plan status so that planning techniques may be used at a finer time scale than has been typical.

CASPER argues for a hierarchical approach to planning wherein the long term plans are developed quite abstractly while shorter term plans are elaborated in more detail, within the same planning model. Ultimately, this blurs the distinction between the planner and executive.

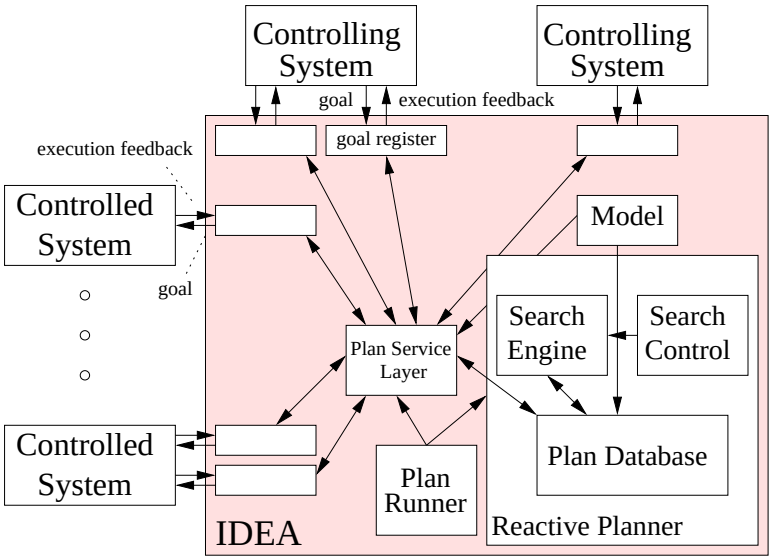

Figure 4.6: The IDEA agent architecture (after Dias et al. 2003).

4.5 IDEA

The Intelligent Distributed Execution Architecture (IDEA) structures the problem of controlling a complex system such as a spacecraft or rover as a set of interacting agents, each with a standard architecture [27, 59]. IDEA contrasts with the usual multilayered approach, wherein each layer has a specialized computational model, programming language etc. The developers of IDEA argue that this lack of uniformity increases the difficulty and expense of development and validation.

The architecture of an IDEA agent is shown in Figure 4.6. External components are either controlling systems or controlled systems. In either case, they communicate with the IDEA agent via *goal registers*, each of which specifies a procedure, inputs, and return status values. These registers cement the "interaction contract" with the external entity, which may be either another IDEA agent or a system that has been wrapped to be compatible.

IDEA agents run on an internal clock. A *plan runner* module is responsible for starting and stopping execution cycles synchronously with that clock as a function of the goals it sees in the agent's goal registers. The *plan service layer* provides an interface between the plan runner and plan database. The plan database is used by the planner to maintain a time line (past and future) for each goal register and state variable.

The IDEA architecture may run with various sorts of planners. In the implementation experiments discussed in [27, 48, 49], a model-based reactive planner, EUROPA, is used as the primary reasoning component. EUROPA is a descendant of the planner/scheduler used in Remote Agent and also uses DDL. Search is controlled with heuristic rules that prioritize subgoals.

In a proof of concept demonstration, the developers of IDEA showed that it could be used to control a K9 rover (similar to JPL's FIDO). The mission scenario involved sending the rover on a tour to photograph a set of targets. Two coupled IDEA agents were used as part of a three-layer hierarchy. A mission-level agent handled goals from ground controllers, passing on lower level goals to a system-level agent, which further decomposed plans for the K9 controllers, and handled monitoring and recovery. An effective 2Hz control rate was reported for this prototype. The authors suggest it may be fruitful to explore compilation schemes so that procedural executives could be induced from agent

models [27].

4.6 Synthesis

Having reviewed some representative current architectures, in this section we extract a picture of common functions that may be candidates for compilation.

Mission Planning and Scheduling From goals, generate a flexible plan consisting of (high-level) actions, perhaps with only limited decisions on resource usage. In Remote Agent, this job is currently done by the Planner Scheduler component (PS) with goals maintained by the Mission Manager. This planner uses backtracking search over a dynamic CSP. TITAN does not directly address mission planning. MDS does not specify a particular solution, but does imply the use of constraint networks as a modeling framework. CASPER introduces the notion of incremental replanning, with heuristic search on a constraint graph to bring the plan into line with changing conditions. IDEA does not stipulate a particular planner, but the EUROPA planner used in experiments is a relative of the Remote Agent planner, based on heuristic search.

Search, with all its promise and dangers, is at the heart of all of these approaches to mission planning. Compilation for the planner may take the form of simplifying the CSP, or other structural analysis of the problem. It might also take the form of limiting the search or of learning search rules offline to speed online search.

Plan Execution Remote Agent's Smart Executive performs run-time expansion and resource management for executing the planned primitive actions, passing them to a real-time controller. Similarly, the TDL-based executive in CLARAty handles the final translation to low-level commands, performs task synchronization, and passes back status information to the planner. The Titan architecture implements a very specific proposal for how to handle interaction with low-level spacecraft systems, in its model-based executive. MDS does not specify a separate executive role, but the implication is that it would be model-based. Likewise, IDEA does not demand an executive per se, but an IDEA agent could (and probably would) be charged with such duties, presumably with a planner of a very different style from EUROPA.

To the extent that we understand them, these executives are already significantly "compiled," as evidenced by their static rendering in procedural languages such as ESL, RMPL, and TDL. To varying degrees, these forms arose from more abstract representations, notably the models used in Titan, though we suspect there is also a respectable amount of offline engineering labor involved in preparing them.

Diagnosis Detecting and diagnosing malfunctions is a critical function for an autonomous spacecraft. Of those discussed above, only the Remote Agent and Titan architectures offer a specific model of diagnosis. Here the mode identification (MI) component of the Livingstone MIR uses techniques of model-based diagnosis to detect and isolate a fault to a particular cause. Using simplified, qualitative models, MI attempts to find the most likely transitions consistent with the observed data. While search is involved in arriving at a diagnosis, the developers note that Livingstone "compiles the models into a propositional constraint network" [60]. MI's combination of an optimized best-first search using unit propagation, and a carefully chosen truth maintenance system (ITMS), met the speed requirements of Remote Agent (tenths of seconds). However, RA's developers also note that they used causal models with "few if any feedback loops". It is unclear that these optimizations

would be applicable or sufficient if richer causal models were required, for example if the feedback could not be eliminated through more careful modeling. The possiblity of compiling a more direct mapping from observables to diagnoses (e.g. connectionist networks) remains interesting for this function.

Reconfiguration Novel reconfiguration of hardware in response to failure is a requirement motivated by many actual mission experiences. In Remote Agent (and by extension, Titan) the mode reconfiguration (MR) component of MIR supplies this function in conjunction with the Remote Agent Smart Executive. The linkage is entailed because a recovery plan for one malfunctioning subsystem may have ramifications for one or more other spacecraft subsystems, as well as long-term consequences that must be weighed carefully, e.g. "pyro valves" can only be used once in the course of a mission. This combination uses both compiled procedures and declarative models. In MR, the Burton planner relies on compilation to avoid runtime search, achieving a constant average case complexity for generating each control action. It achieves compactness by constructing a set of concurrent policies (rather than a single policy for the whole problem space). However, it should be noted that it only considers *reversible* actions, deferring the irreversible ones to a higher level reasoner, or perhaps a human executive.

Vehicle Coordination NASA is increasingly moving in the direction of autonomous or semi-autonomous multi-satellite constellations for a variety of purposes, for example earth observing, VLBI (very long baseline interferometry), or coordinated observations of the earth's magnetosphere. There has been a fair amount of design work on control and coordination architectures for spacecraft constellations, less on the high-level planning and execution that falls within the scope of this study (though see, for example [19]), and little or no experience with deploying such systems.

At the current stage of development, we are not yet able to identify functions that are unique to this kind of multi-platform autonomy, qualitatively different from the functions already identified, that would be promising candidates for compilation. This is not to say that satellite constellations would not benefit from an increased use of compiled automation. The proposed deployment of several different constellation-based missions strongly supports the need for increased autonomy in the future, potentially in very limited hardware (e.g., earth-observing nanosats), all of which serves to strengthen the motivations for compiled automation.

Chapter 5

Methods

In this chapter, we describe various tactics that can be applied to implement compiled automation. Many of these have potential application across a wide range of functions, depending on mission parameters. While not attempting to provide a taxonomy of compilation, this chapter will suggest, through examples, some promising categories of compilation that have been or could be used to enable autonomy in architectures like those considered in Chapter 4. The compilation "tactics" considered here include:

Approximation – working with a similar but easier problem

Reactive Methods – using a more direct input/output relation

Learning – learning policies that can be applied rapidly

Indirection – coupling a fast system with a slower one

Speeding Search – via limiting and search control

These tactical categories may be applied in combination (implementing an approximate policy, for example). For any given function, it can easily be the case that more than one approach might prove useful, alone or in combination.

5.1 Approximation

The gist of the Approximation tactic is to transform some part of the domain theory to an approximate form that can be exercised very efficiently in operation. It typically entails accepting some loss of coverage or precision when compared to what could be computed with the full theory, given sufficient resources. The approximation tactic may be coupled with some other method as a fallback to catch when it fails.

One example of an approximation strategy is described by Selman and Kautz [77]. They compile a knowledge base to an approximate theory (based exclusively on Horn clauses) that can be queried in linear time. Because some concepts cannot be expressed precisely in the approximate theory, a tradeoff in precision or coverage was accepted: sometimes the query would be answered by "I don't

know", which would demand solution by slower methods. This will be a very good trade in certain applications.

A completely different form of approximation is employed by connectionist models, e.g. artifical neural networks, [46]. These numerical methods approximate a complex functional relationship between variables. The variables may be discrete or continuous. Certain forms of these models can be shown to return an answer in constant time. They can be trained against simulations or empirical data to produce the desired solution given certain conditions. In some cases, such models can even be more space efficient than the original tabular form of the data, for example in approximating a smooth manifold. Difficulties in training and generalization with such models may be a limiting factor.

Use of approximation as part of a compilation process is described by Dearden et al. in an application of contingency planning to planetary rovers in [25]. Computing the utility to be gained by inserting a contingent branch in a plan is an intractable problem, but an approximation that propagates utility distributions backwards through a plan-graph like structure appears to practical.

Bayesian belief networks as a model of reasoning about causation under uncertainty may be used for functions like diagnosis or to estimate the likely outcomes of a plan [69]. The complexity of operations on these networks has prompted various forms of approximation, notably the use of qualitative (order of magnitude) representations such as System-Z^+ [42]. Such systems can work with a finite set of belief values, replacing floating point arithmetic with operations like minimization. The use of particle filter techniques, an application of Markov chain Monte Carlo methods, for approximate Bayesian inference is being pursued by Dearden et al. at NASA Ames for probabilistic fault detection in systems like rovers [24].

5.2 Reactive Methods

Reactive methods are already incorporated to varying degrees in the autonomy architectures we have discussed. The reactive tactic largely does away with search in favor of a more direct mapping from situations to actions. In essence, a control plan tailored for the domain (as opposed to a single problem instance) is constructed offline. The resulting plan can be efficiently executed, and ideally handles a wide variety of common situations.

While the concept is in some senses as old as automatic control, its popularity in artificial intelligence saw a resurgence with the work of Agre and Chapman on "Pengi" a program that plays a computer game using such a strategy [1] and Rosenschein's work on Situated Automata [76]. The work on representation-free intelligent agents by Brooks advances a similar approach with his subsumption architecture [16]. Drummond and Bresina's work on Situated Control Rules was among the first to compile reactive planners from a domain model and a goal specification [28].

There is some debate as to whether and how this can be done efficiently and in a compact way, e.g. [39]. Muscettola et al. warn:

> 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" ([60], p11.)

Arguably, handcrafting reactive plans is not fundamentally different from current practice for specialized flight software. The current state of the art relies on engineers making simplifying as-

sumptions to make the domain tractable. Modern procedural languages in use in the architectures discused in Chapter 4 like PRS, RMPL, RAPS and ESL offer advanced ways to express reactive plans [55, 51, 36, 38].

There is a continuum between policies where every step in a plan is conditioned on the current state, and a "conformant plan" wherein each step follows the previous one with no reference to the evolving state of the world beyond what was conceived by the planner at the time the plan was generated. *Contingency planning* occupies a middle ground, assuming that the planner's knowledge of the world is incomplete, but that sensor information can be obtained that provides information on the world state as the plan executes [10, 54, 25]. The contingent plan, with embedded sensing actions, can respond to the new information without replanning, since the contingency had been considered offline. In this way, contingent planners may offer a way to produce an essentially complete policy from a more declarative input specification, viewing it as a nondeterministic control problem over a belief space. Bonet and Geffner's GPT offers an integrated framework for dealing with uncertainty and partial information [11] through contingency planning. Whether contingent planners can be made efficient enough to generate comprehensive contingent plans for realistic examples is an open research question. Published examples have been relatively small problems.

5.3 Learning Reactive Plans

While there are many ways to produce reactive plans, one of the most intriguing is via learning. There are several research threads in this broader theme that deserve mention. Khardon has suggested that reactive plans can be learned in a supervised manner [50]. In his L2ACT system, a set of solved instances randomly sampled from a family of planning problems was presented to the learner. The training solutions were generated by GraphPlan [8]. The learner used a decision-list learning algorithm ([75] to produce a policy that emulated the decisions generated by GraphPlan. Policies (called "production rule strategies") are ordered lists of existentially quantified rules. The learner considers all candidate rules and iteratively selects the "best" until coverage is achieved. The learned policies were then tested on new instances of problems from the same domain. In logistics and blocks world problem domains, the learned policies were able to solve an appreciable number of the new problem instances. Larger problems could be solved after training with smaller instances, and some robustness was illustrated, e.g. dealing with examples where no learned strategy offered a precise match. This technique appears to be applicable to domains beyond the classical ones examined, notably stochastic ones where actions have probabilistic effect.

Extrapolating from the work of Khardon, Martín and Geffner [53] have demonstrated an enhanced method of learning what they call "generalized policies" by using *concept languages*. Khardon's original formulation relied on the user defining useful "support predicates", useful compositions of primitive predicates, that may be difficult to generate in a poorly understood domain. Concept languages (a.k.a. description logics) provide a natural way to express complex concepts from primitive concepts and roles. Martín and Geffner used a concept subspace to let the learner search for its own support concepts. For example, in a blocks world domain, the concept:

$$(\forall on_g^*.(on_g = on_s)) \wedge (\forall on_g.clear_s)$$

which might be denoted NEXT-NEEDED-BLOCK was found; this captures the notion of block whose destination block is clear and "well placed", i.e. where it should be in the goal state.

Using a training set of 638 randomly generated blocks world problems involving 5 blocks, solved by the HSP planner [10], their Lisp prototype was able to produce a generalized policy over the 216,684 candidate concepts in several hours. The resulting generalized policy solved 96% of new 5 block problems, 87% of new 10 block problems and 72% of new 20 block problems.

A related approach for using offline learning to create a reactive online policy is the work of Fern, Yoon and Givan [3], who explore the idea of *approximate policy iteration* (API). This technique attempts to find policies that can be applied to any problem in a given domain. This is done in two phases. In the first phase, the STRIPS/ADL planning problem is "rolled out" into a Markov Decision Problem (MDP) in which each MDP state is a planning problem instance, i.e. an initial state and a goal. Using a training set of Monte Carlo samples, each state is labeled with the associated Q-costs for each action (not just the best action). The overall algorithm can be viewed as a kind of approximation to policy iteration, hence the name API.

In a second phase, the learning algorithm attempts to improve on the initial policy π by selecting a new policy π' that minimizes the estimated Q-cost. The learning algorithm uses a concept description language similar to that of [53], incrementally constructing a decision list of bounded-size rules. The language imposes a kind of bias on the learner that is advantageous. For example, in a logistics domain, the concept language may encourage discovery of rules such as "unload any object that is at its destination." Lagoudakis and Parr have also explored the use of this policy bias approach [52].

One thread of research directly addresses the uncertainty in the mission using the perpsective of Markov Decision Problems (MDPs) and partially observable Markov Decision Problems (POMDPs) [13] and [10]. Zilberstein, Washington, Bernstein and Mouaddib apply decision-theoretic methods to planning the science missions of planetary rovers [7, 86]. Planetary rovers face a level of uncertainty significantly higher than typical probes. Their aim is to provide the highest information return for a set of targets, while respecting resource and time constraints. To avoid having the rover attempt to solve difficult MDPs in situ, and likewise to avoid having to send a possibly large policy file to the rover over a limited communications channel, they use an offline analysis of each activity the rover can perform to compute "precompiled policies", taking advantage of the nearly independent nature of the subprocesses represented by each activity (weakly coupled MDPs). The precompiled policies need to be *adapted* at run-time to reflect the remaining tasks and resources, referred to as "opportunity cost". In their solution, an efficient approximation of the opportunity cost is computed offline; this is then used to generate compiled policies for different levels of opportunity costs for each activity. The set of compiled policies is loaded on the rover, which will then have compiled decision rules which should be good for a range of eventualities.

Reinforcement learning with weakly coupled MDPs was proposed by [6] for planetary rover control as an alternative use of the MDP formulation to the adaptive planning technique discussed above [86]. The learning approach does not need to know the exact model of the environment. The authors further mention that the learning approach could be used on the rover to refine a compiled policy to better match the actual environment. The scalability of this approach is yet to be determined.

Evolutionary methods provide yet another methodology for learning reactive plans. In an offline setting, the learner would interact extensively with a simulated environment that could introduce various challenges for the system, and a cost function to evaluate the performance of a policy. Grefenstette and Schulz describe the SAMUEL system [43], which successfully evolves control plans in various domains. SAMUEL actually offers a hybrid learning approach, with evolutionary steps (recombining and mutating policies) interpersed with reinforcement learning as the policy interacts with a simulated environment. A survey paper by Moriarity, Schulz and Grefenstette [57] provides more perspectives on this approach.

5.4 Indirection

The Indirection tactic is to couple a subsystem that can perform rapidly but with limited scope, with a slower but more comprehensive subsytem that can reprogram the former on-line, given changing

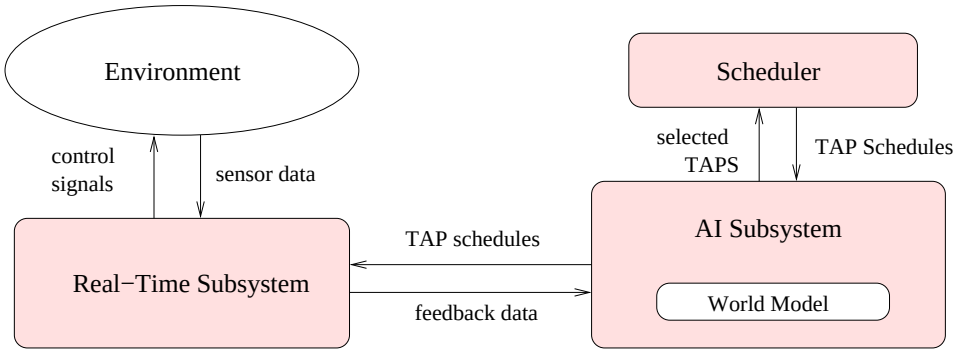

Figure 5.1: The Cooperative Intelligent Real-time Control Architecture, CIRCA, (after Musliner, Durfee, and Shin 1995).

conditions. One of the best developed of these methods is CIRCA, the Cooperative Intelligent Real-time Control Architecture. CIRCA combines sophisticated planning with real-time control guarantees [63, 62]. The CIRCA architecture is shown in Figure 5.1. Two major components run in parallel: a real-time subsystem that has time-predictable control responses, and an AI planning system that performs high level reasoning, coupled with a scheduler, to generate the low level control plans for the real-time subsystem. The control plans consist of schedules of test-action pairs (TAPs), each a production rule annotated with resource and timing requirements. Intuitively, the AI subsystem focusses on "task-level" goals, while the real-time subsystem drives towards "control level" goals. In this way, CIRCA attempts to be intelligent *about* real-time as opposed to intelligent *in* real-time.

CIRCA automatically classifies task-level and control-level goals by analyzing a domain model and priorities specified by the user. Task level goals are decomposed into subgoals, and control plans are built for each subgoal that avoid failure. CIRCA's world model includes explicit world states and transitions. States represent complete descriptions of the world. A special "failure" state subsumes all states that violate critical constraints. Transitions may arise from actions that the system takes, or from external events or the passage of time. A single token represents the "current" state at any given time. CIRCA relies on state enumeration; this allows it to prove plan correctness yet can be a limiting factor for complex domains.

Scalability in CIRCA has been addressed by introducing abstraction. The *dynamic abstraction planning* (DAP) extensions to CIRCA are discussed in [40]. DAP provides abstraction in the description of *states*–instead of being full world descriptions, only some features are included initially. As further details are required, the abstract state can be "split" as needed. Unlike other approaches to abstraction, the planner itself decides which features to abstract dynamically, and features that are abstracted away in one part of the plan may become explicit in another.

An application of DAP-CIRCA to spacecraft planning is discussed in [61]. CIRCA was shown to solve a prepositioning problem similar to one that arises in missions such as Cassini Saturn. To solve this problem, the prudent planner must recognize that a backup inertial reference unit (IRU) must be preheated far in advance of an engine burn to prepare for contingencies with the primary unit and thus avert mission failure.

Using CIRCA in the negotiation of coordinated plans is discussed in [64], suggesting an application to satellite imaging coordination. Decision-theoretic extensions of CIRCA are suggested in [44] which uses sampling of a stochastic simulation to estimate choice utility. Model-checking extensions of CIRCA are discussed in [41].

5.5 Speeding Search

Search is a common factor in several of the functions needed for autonomy. So, what kinds of techniques can be employed to make search more predictable or more efficient, assuming it is not simply dispensed with through one of the tactics above? Two broad classes of possible approaches are finding effective heuristics, and limiting the search space explored, even in the worst case. A third approach we have already mentioned is to compile the search space into a form that can be searched especially rapidly. Sytems like Remote Agent MIR and Titan compile the original problem to a constraint network that significantly simplifies the job of online search. In Livingstone, a conflict database caches "nogoods"–incompatible sets of variables assignments–using a truth maintenance system. This may also be construed as an application of learning since it avoids the need to rediscover these conflicts [68]. There is substantial recent work within the AI Planning community, as well, on analysis and compilation of a domain model so as to make search more efficient (see, for example [8, 37, 30]).

Considerable research in this vein has been conducted on the topic of knowledge base compilation (for a review see [22]). The tactic here is to take a propositional theory and to compile it (perhaps quite laboriously) to a form that can be used to answer certain classes of queries online in polynomial time. This tactic has been demonstrated in relevant applications in Darwiche's use of decomposable negation normal forms (DNNFs) in diagnosis [21]. A diagnostic model can be reduced (compiled) into two components: one to condition NNFs and other to extract minimal instantiations (diagnoses). Both components offer opportunities for runtime efficiency not present in the original specification. Analagous techniques can be applied to planning.

In related work, Darwiche and Provan show how Bayesian belief networks can be compiled (via clustering, pruning and conditioning) into arithmetic expressions called "Query DAGs" [23]. Answering queries becomes a simple matter of evaluation, saving the cost of traditional belief net inference algorithm. Moreover, the space complexity of the online algorithm is no worse than that of the standard algorithm.

Offline compilation of heuristics for controlling search is a further form of leveraging offline computing resources, though not one that will result in performance guarantees. The Remote Agent planner included mechanisms for guided backtracking using an "approximate oracle" for search control, essentially incorporating well-engineered heuristics [49]. Learning control knowledge for planning has also been discussed by Aler, Borrajo and Isasi [4], and by Estlin and Mooney [34]. This work can support planners involved nearly anywhere along the spectrum of compiled autonomy.

The PRODIGY architecture has been used to study several techniques for learning to plan [80]. One such approach is to learn control rules via a static analysis of the problem using an explanation based learning (EBL) algorithm. This can be done without specific instances, or by using the results of training problems to focus the EBL algorithm. A derivational analogy approach uses justifications built during the solution of (training or actual) problems to reason by analogy to future similar problems. Some of the work on learning for planning in rovers by Estlin et al. [32] is related.

5.6 Summary

This chapter has described categories of tactics for compiling the critical functions that occur in the autonomy architectures of Chapter 4. The space of possible tactics is certainly larger, e.g. we have not really touched on the synergy of functional components coupled by shared compiled models. Continued research is likely to reveal still further methods to transform reasoning systems

into forms well suited for spacecraft. Chapter 7 will address how these techniques could work in action. Chapter 6 will examine tradeoffs in the application of the techniques to different spacecraft functions.

Chapter 6

Implementation Tradeoff Issues

What criteria must be weighed when evaluating potential applications of compilation for autonomous spacecraft? At least four broad criteria are entailed: Cost, Schedule, Performance, and Safety.

6.1 Cost

Cost refers to the entire life cycle cost of the mission, including development, launch, and operations. Development includes generating the mission objectives and requirements, designing the mission profile and component vehicles, detailed design and construction of subsystems, integration into systems, and testing throughout the process. Launch is the act of getting the spacecraft off of Earth and on a trajectory to its final objective.

The remaining portion of the mission falls into operations. This is usually the largest cost component of a mission, often far exceeding the development cost for the spacecraft. It is important to consider the impact that mission development has on the operational costs. A well designed and tested spacecraft is more likely to meet its operational cost estimates than one in which a design error slipped past the development team. Furthermore, a spacecraft designed with flexibility in mind will be less expensive to adapt when (not if) failures or opportunistic goals appear.

Automation can effectively reduce the operational component of mission cost, through reducing the amount of manual intervention required for mission operations and monitoring. Wertz [83] describes reduced operations costs as a key benefit to using autonomous navigation and orbit control. [71] makes similar claims for a procedure automation approach. An important caveat to this is that the mission team must trust the automation (compiled or otherwise) and understand its limits [72].

Compiled automation techniques play a role here in that the compiled artifact may engender greater trust either through greater verifiability or other demonstrated virtues. For instance, there clearly was a benefit to automating the management of the Galileo tape recorder (see section 2.2). Through the "compilation" of the required changes into a set of sequences and configuration changes, they could be verified by the mission team and the mission team would trust the steps to be executed as designed. It is unlikely that the creation of a reasoning engine to operate the tape recorder would have been supported. Finally, communications also contributes to mission cost. Reducing the amount of information required to operate a spacecraft can yield savings.

6.2 Schedule

Schedule refers to the timeline that the mission and spacecraft must follow to be successful.

6.2.1 Mission-level Schedule

The mission-level (or strategic) schedule considerations include: Design and Development, Assembly, Test, and Launch Operation (ATLO), Cruise to Target(s), Science at Target(s), and End of Mission.

From a hardware perspective, Design and Development is driven by the ATLO schedule. If the hardware is not on-board when the vehicle is launched it will probably never be on board (some exceptions include Earth orbiting vehicles, space stations, etc. that can be upgraded, e.g., the Hubble Space telescope repair). For software, however, the ATLO schedule *can be made* less of a factor, with the Science at Target schedule being a larger factor. As the Galileo experience pointed out, for missions where a long cruise period is planned, software can be and often is modified in flight to react to problems, improve performance, and increase functionality of the spacecraft. In the case of Galileo, for example, nearly all of the software was modified before the conclusion of the mission, with many changes made between each science pass. These modifications were mostly reactions to unforeseen problems, and required extraordinary innovation and effort on the part of the mission staff. In other cases, the upgrades were a reaction to knowledge gained after the mission was launched or to take advantage of opportunistic science, e.g., data regarding the Jovian environment collected by Ulysses was incorporated into Galileo's science instruments and, as another example, NEAR-Shoemaker's gamma-ray spectrometer software was redesigned to collect data from only 4 inches from the asteroid Eros' surface after the mission was extended to achieve this remarkable science goal[66].

In the case of automation, tradeoffs exist regarding how much of the automation must be implemented before launch versus what could wait until after launch. Of course, this relates back to the issues of trust, but it is certainly the case that core automation functions (e.g., basic navigation and housekeeping) could be present and validated at launch time, while other functions could be refined after launch.

As an extreme case, a deep space mission could be launched with the necessary computational hardware but with only the software necessary for it to get from Earth to its destination, with the rest of the system to be uploaded during the cruise phase. Landers and surface vehicles could likewise be programmed after launch. This increases costs in terms of DSN time to transmit the software, stretches development past launch, and may increase perceived risk. The benefit is a lengthened schedule for designing, implementing, and testing the software (several years in the case of a mission to Jupiter). In addition, the post-launch software development could reflect exactly the hardware that was launched and proven operational. The net effect may be to reduce risk and life cycle costs.

A balance would need to be struck between software that is flight ready prior to launch versus after launch. In some cases, compiled automation may provide the means to take advantage of this. At a minimum, compilation approaches provide support for making the system modular, so that the independent components can be verified and tested. Thereby, new uploads could be relied upon not to interfere with the core functions already on-board.

6.2.2 Spacecraft-level Schedule

The spacecraft-level schedule refers to planning and scheduling the day-to-day operations of the spacecraft and is concerned with a much smaller granularity of time than the mission-level schedule. The spacecraft schedule must continuously interleave housekeeping tasks (navigation and attitude control, telemetry, battery charging, built-in-test, etc.) with science tasks (data collection, processing, transmission potentially from different combinations of the numerous instruments). Tools such as the NASA Ames MAPGEN Planner are helping the operations team to more accurately and quickly create these schedules and define contingency plans if the primary schedule is not followed precisely.

In the past, such schedules were based on timed commands [45]. If an operation took a few seconds or minutes longer than the window allowed, a significant chunk of science could be lost while the schedule was adapted. Recent work on sequence execution adds sequence execution based on events, incorporates planners and schedulers that automate portions of the sequence generation process, and adds robustness to sequence execution [78].

As a team gains experience with a mission and the operations become routine, the potential exists for some scheduling chores to be compiled into reusable components or policies. Ideally, day-to-day operations will become progressively more automated as the experience of the mission team is compiled into terrestrial systems or onto the spacecraft itself.

6.3 Performance

Every spacecraft has finite resources available for collecting and processing data, operating the spacecraft, and deciding what to do next. Past missions have generally had severely limited onboard resources, and so most mission-level processing (e.g., figuring out the details of an orbital pass or the next motions for a rover to take) was perforce accomplished on the ground. While the available processing power and memory grows steadily over time, so does mission complexity. Furthermore, high-radiation environments impose their own physical constraints. Consequently, as computing hardware elements shrink and become more densely packed, the effects of radiation will be harder to mitigate, possibly resulting in a practical upper bound for the benefit gained from Moore's Law. Add to this the strong pressure to migrate additional functionality onboard in the process of increasing spacecraft autonomy, and resource limitations are likely to be an issue for the foreseeable future.

After launch, software can generally be upgraded, but hardware usually cannot. Budgeting a reserve measure of processing power and storage (with the power to run it) to support post-launch upgrades of compiled functions, should be part of the design tradeoff equation. Galileo's onboard computing consisted of six 8-bit microprocessors with 384Kb of memory in the data processor. The attitude and articulation control subsystem (AACS) had a 16-bit computer, but only 64Kbytes of memory, nearly all of which had to remain unaltered. Science instruments were controlled by a single 8-bit microprocessor with very limited memory. In addition to the compression software, seven of the instrument teams had to rewrite the control software for their instruments (other instruments either lacked the resources or could not be reprogrammed) [45]. Things might have been considerably easier for the Galileo operations team if the hardware had included a spare processor and extra memory.

Compilation can help address these issues, first by the simple fact of reducing the resources needed, and second, and more importantly, by providing the framework necessary to make principled tradeoffs among the various uses to which those resources might be put.

6.4 Safety

Safety of a mission or spacecraft is a multi-faceted issue that includes everything from principled design methodologies (e.g., configuration management, regression testing, etc.) to formal V&Vof the system, to feedback to the mission team on spacecraft status, to providing explanations of spacecraft actions, current state, and intended actions. A compiled approach has the potential for increasing overall safety of the automation and, consequently, the spacecraft, in cases where the compiled form is easier to verify and test, or can be shown to operate more predictably. We review some points related to safety that should be considered.

Past missions have shown that validating and verifying systems has an enormous impact on the success or failure of the mission. For example:

- The Mars Climate Orbiter failed, simply because one subsystem used metric units of measurement while another used English [65].
- In 1988, an error in a software upload to Phobos 1 was blamed for deactivating the attitude thrusters, resulting in a loss of lock on the Sun and the batteries being depleted, resulting in a loss of the spacecraft [82].
- On Galileo, the command and data subsystem safed itself because a software error caused an overflow of commands in a buffer, causing a processor to exceed the allotted time on a given task. Software was uploaded to the spacecraft to handle this overflow more gracefully.

With respect to automated systems, safety demands that the automation is represented faithfully and in sufficient detail that it can be analyzed for correctness. Generate and test verification approaches, as is common with knowledge-based reasoners, have not proved reliable. Analysis of simpler compiled artifacts (e.g. policies or reactive plans), can provide guarantees that are not available from the original implementation.

The need to reverify all of the automation for every software change should be avoided. Also, the automation should be tied to the set of design characteristics upon which it is based so that as those change the impact can be seen. Compilation can support both of these desiderata, in the first case by supporting the addition of modules to a common architecture in ways that preserve necessary properties (e.g., latency or other intermodule timing constraints) so as to preserve verification, and in the second case by mapping from a source form that captures the relevant design information.

6.5 Tools for Making Tradeoffs

Local tradeoffs (within a single function, basically) may be conducted by using rigorous stochastic or decision-theoretic methods, but at the level of mission definition or spacecraft design, tradeoffs will commonly be informed by subjective estimates of priority, risk, reliability, and cost. A host of experimental and statistical analysis tools and techniques have been developed to assist with these tradeoffs and to assess the baseline capability of a process, diagnose the most significant sources of error or variation, and then measure and analyze the impact of improvement implementations, e.g., [12, 14, 56]. The widely publicized Six Sigma and Design for Six Sigma programs implemented at major U.S. businesses like Motorola, General Electric, Honeywell, 3M, and others combine several of these tools into a process improvement methodology. A detailed discussion of these tools is out of scope of this study (see, for example, [20] for more information), but one tool, the Quality Function

Deployment (QFD), with some potential of helping to structure tradeoff discussions, is introduced as an example.

The QFD is a mechanism for transforming knowledge about customers' prioritized needs or requirements into focused action plans. By building on the collective knowledge of all stake-holders (i.e., the suppliers, mission operations, scientists, funders, domain experts, etc.) and using weighted-stake-holder-needs, alternative features or solutions can be scored based on the needs met, how well it meets them now, and what effort (or cost) is required for an improved result (meeting a new need, or meeting a need more completely). The artifact of the process is a matrix capturing these assumptions that can be revisited as more information becomes known or the situation changes.

The QFD is particularly useful when resources are limited, the selection decision criteria are abundant, and the relationships between needs and solutions are complex. Although QFD traditionally has been applied to new products or product improvement programs, space missions also share the features where a QFD is useful. Determining whether and how the QFD or some related tool could benefit space mission design is a topic of further study. The work done by Proud et al [72], for example, might be another starting point in that they developed a ranking approach for trust in automation. By adding information about what would be necessary to raise the collective trust levels (i.e., the 'cost' of improving trust), it would become a more useful tool for guiding tradeoffs.

Chapter 7

Case Studies: Application of Compiled Automation

The previous chapters have looked at functional requirements of space missions and have introduced several architectures and methods available not only for automating some of the functions, but also for delivery in a more compiled form. In deciding on the appropriate approach to future missions, the tradeoffs discussed in Chapter 6 need to be considered for both an approach to automation in general and whether or not that automation should (or could) be compiled.

This chapter examines two different future space missions. The first is the Jupiter Icy Moons Orbiter (JIMO) and the second is a proposed Venus lander.[1] These were selected because of interest to NASA and because they are qualitatively very different missions.

7.1 Jupiter Icy Moons Orbiter

The Jupiter Icy Moons Orbiter (JIMO) is a proposed mission with the objective to orbit three planet-sized moons of Jupiter (Callisto, Ganymede and Europa). These "Icy Moons" have been a priority for detailed exploration ever since the Galileo mission found evidence for subsurface oceans, a finding that ranks among the major scientific discoveries of the Space Age [67].

In addition to targeting important science objectives, JIMO would raise NASA's capability for space exploration by demonstrating safe and reliable use of electric propulsion powered by a nuclear fission reactor. This technology not only makes the JIMO mission possible, but will do so with more than ten kilowatts of electrical power; a much larger supply of electricity than previous deep space missions. With a planned data rate of 10Mbps (at Jupiter), JIMO will achieve an expected data volume of more than 50,000 GBits over JIMO's mission. With planned continuous observations and communications, JIMO's science instruments will be more powerful than ever before and the scientific information captured will be unprecedented [17].

The JIMO mission is not without challenges, however, and some of these should be addressed with automation. To understand where compiled approaches to that automation would be most effective, we consider some specific JIMO functions. From a mission-schedule perspective, some interesting

[1] One concept under development for this mission is called Venus SAGE, after the earth observing missions of the same name.

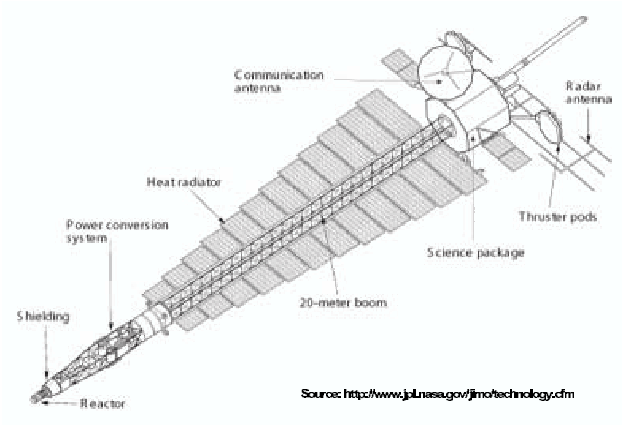

Figure 7.1: Artist's Conception of JIMO Spacecraft, from [67].

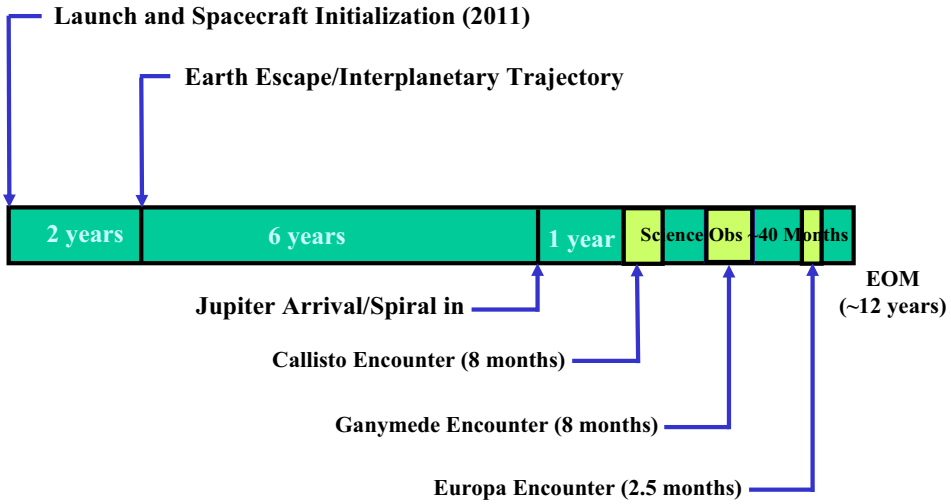

Figure 7.2: Pre-Decisional JIMO flight timeline, from [67].

conclusions can be reached from the pre-decisional data provided in [17]. The target launch date of JIMO is 2011, with a nearly 9 year journey to Jupiter (see Figure 7.2).

In this case, the time en route to Jupiter is approximately two years longer than the time remaining before launch of the spacecraft. The opportunity exists, therefore, to exploit this time by making a deliberate decision to split the JIMO development into two distinct phases. The first phase, covering the period up to launch, would concentrate on the hardware components of the spacecraft and those software functions required for the launch and cruise phase of the mission. The second phase, covering the period after launch up to arrival at Jupiter, would focus on the additional software functions required for use at Jupiter, for example software that processes the science data. If properly executed, this strategy more than doubles the time available to develop, test, and validate the software that will be operating JIMO as it tours the icy moons of Jupiter.

A study would have to be conducted to determine what should exist pre-launch and what might be developed post-launch, but the key will be to provide an architecture that enables the addition (and V&V) of software added after launch not only to the spacecraft but also to the various testbeds being used for development and testing. It is extremely important that software uploads after launch are seamlessly folded into the existing software to incrementally improve functionality and not introduce faults. Further, the new software added should not require a reverification of all of the existing software.

Compiled automation can satisfy these requirements through the generation of new software as an artifact, or a set of artifacts, that can be independently verified on the ground prior to being uploaded to the spacecraft. For example, consider the System Health Management (SHM) function of JIMO. Clearly, JIMO will need an SHM capability immediately after launch to manage failures that may occur during cruise to Jupiter, since the engines and certain other equipment will be operating the entire way there. SHM activities that pertain to operations of the science instruments may not be required until that equipment is in operation. As these components are created, they can be compiled into a fault table, for example, in such a way that the changes can be verified automatically, and the spacecraft-based algorithm that operates on that table can function without requiring additional V&V.

This split development schedule can also facilitate V&V. After launch, the hardware and software included on the spacecraft will be known with certainty, so that the high fidelity testbeds can be

updated and, possibly, even replicated. Lower fidelity testbeds can also be revised to more accurately reflect the launch configuration.

Another component of the mission schedule is the spacecraft-level schedule, i.e., those functions that occur on the spacecraft on a day-to-day basis. The round trip light time (RTLT) to Jupiter and back is nearly two hours. Hence, there cannot be any expectation of fine-grained control of JIMO from Earth. The expected data rates and data volume generated by the science instruments together with the operational complexity of the orbital passes (lots of things to do) will make business-as-usual mission operations (e.g., coding detailed timing instructions and uploading to the spacecraft as was done on Galileo) both extremely difficult and detrimental to mission objectives.

Consequently, JIMO will require as much automation as possible to handle day to day tasks (e.g., attitude control, backup power maintenance, etc.). These tasks, however, are also mission critical and although automation is an attractive option, that automation must be verified to the level of other flight software. While non-deterministic reasoners may be involved, it is unlikely that they will directly be controlling the spacecraft. Rather, the outputs of the reasoners will be executed and, consequently, these outputs will need to be verifiable. Further, since these tasks will be as close to routine as anything on JIMO, it is likely that these compiled outputs will be executed over and over again as certain conditions arise, so that the effort required to generate, verify, and validate the compiled artifacts will be amortized across the life of the mission.

Another significant driver of space missions is cost. In general, for long duration missions, such as the proposed JIMO mission, the cost of the operational tail exceeds development and launch costs, especially if the mission encounters problems. A key component of the operational cost is the planning and execution of the science mission. Compiled automation techniques can be utilized to reduce this aspect of the mission costs in the following ways:

- As the mission planning team gains more experience in operating JIMO, provide the means to capture that experience and compile it into the automation on the spacecraft. For example, if the science team finds itself always planning the same sequence of events, provide tools to facilitate the preparation of a compiled set of commands that accomplish those objectives.
- Provide on-board automation that accomplishes the routine navigation, attitude adjustment, command handling, and other housekeeping tasks of JIMO. This will relieve the mission team from worrying about these tasks, but it must be done in a deterministic manner so that real-time and mission safety guarantees can be satisfied.
- While not strictly related to compiled automation, it may make sense to consider including computational hardware well beyond what is minimally required for operations in both the science instruments and in the greater spacecraft. For JIMO, the power budget is not a strongly limiting factor, and the additional hardware will provide flexibility and resources to the mission and science teams to more easily react to future problems and opportunities. While this will increase pre-launch costs, it will very likely reduce long-term costs and/or improve the quality and quantity of data collected during operations. As discussed in Section 2.2, the mission team for Galileo on several occasions found unforeseen uses for existing computational and storage hardware.

JIMO will command more electrical power and communications bandwidth than any other deep space mission to date. This gives JIMO an opportunity to collect, process, and download huge amounts of data to Earth for exploitation. However the ambitious science goals for this mission will require the science instruments to rise to the challenge and make heavy use of those resources, especially communications. Although 10Mbps seems like a lot of data, it is not large in comparison with Earth orbiting satellites which can individually have data rates an order of magnitude

larger, e.g., Radarsat-1 has a 105 Mbps data rate ([79]). JIMO's science instruments are projected to include a Payload Data System with redundant computers, Near, Medium, and Wide Area Cameras, InfraRed Spectrometer and Thermal Imager on a scan platform, Laser altimeter, Ion Counter, Particle Detector, Mass Spectrometer, Dust Detector, and Plasma Spectrometer on Turntable allowing 360-degree scans, and a boom-mounted Magnetometer [17]. Each of these instruments will be more capable than any of their type that have gone to space before. Nor is it beyond the realm of possibility that some kind of equipment failure will substantially reduce the power, bandwidth, or computational capacity available on-board the spacecraft. Planning and scheduling of resources will be as important a capability for JIMO as on previous missions.

JIMO safety revolves around system health management, V&V, and the system design methodology. These each have been touched upon already in the context of other tradeoffs. Clearly, mission safety will be of paramount concern, not only because JIMO will have a nuclear reactor on board, but also because it will be exploring an extremely harsh environment for a long period of time. With a RTLT of nearly two hours, any action that must operate within that time window will need to be handled by the spacecraft, including reactions to failures. In past missions, the spacecraft would safe itself when it encountered a problem it could not solve itself. These sorts of delays caused loss of days of science data, e.g., loss of key data from an encounter with Europa on the Galileo mission ([45],p. 226). Furthermore, JIMO's ion propulsion system differs from chemical rockets in providing a continuous but low-level thrust and without any capability to execute high-impulse correctional burns. Consequently, in the case of JIMO and missions like it, if the propulsion system is turned off for some period of time, as commonly happens when a spacecraft safes itself, the spacecraft may rapidly drift off course because of the significant gravitational forces present in the Jovian environment, past the propulsion system's ability to return the spacecraft to its original course.

For JIMO, technology and on-board processing are sufficient that a much larger range of faults should be able to be detected, even predicted, isolated, and automatically recovered. Of course, to be qualified for space operations, the actions taken will need to be verified to not hurt the spacecraft, to react quickly enough to mitigate the failure effects, and to communicate the actions taken to the mission team so they can maintain situation awareness. In several cases, these reactions may be developed over time, even during cruise to Jupiter, as the mission team gains experience with the spacecraft. Thus, the previously-discussed approach of off-board compilation, automated verification, and uplinking is a good fit here as well.

Design and V&V methodologies and tools need to not only consider the long-term during pre-launch, but also to maintain this discipline and rigor throughout the mission. For example, in addition to the high gain antenna failure early on, the Galileo spacecraft experienced several radiation-induced faults due to the Jovian system's radiation belts and strong magnetosphere, sustained numerous micro particle hits, and had a recurring failure because of a build up of fine particulate matter (from degradation of the slip rings) on the electrical connectors on the spin bearing assembly [45]. Similar effects should be expected for JIMO, in addition, the proposed JIMO spacecraft is itself expected to generate significant local environmental effects. For example, the power distribution system, ion thrusters, and solar arrays will generate large magnetic fields, and the Ka band communications system and proposed ice-penetrating radar will produce large radiated E-fields [17]. Consequently, JIMO can be expected to encounter failures which may not be fully foreseeable during system design.

As the software on the spacecraft evolves with the mission, it will be crucial to easily, and, ideally, automatically, determine which modules are affected by a particular software upload to JIMO and which automation components are either no longer valid or also require update. This requires a principled approach to a modular architecture, with well constructed interfaces between each module, including relationships to the control/automation software for those modules. If reasoning systems are used to implement parts of the automation, the base models will need to be analyzable

to determine this information. Revalidating the entire set of spacecraft models for each software update will quickly become intractable.

When possible, therefore, the knowledge bases used for automation on the spacecraft should also be modular and governed by a set of well defined interfaces between each module. The outputs of these knowledge bases will also need to maintain a lineage back to the models that defined them. For example, for each compiled artifact, it would be useful to maintain a mapping back to the reasoning (i.e., the assumptions) that was used to create that artifact. This mapping would make possible a complete review of existing spacecraft automation to identify those components that may require modification to utilize the new software updates.

7.2 Proposed Venus Lander

In contrast with the JIMO mission, the proposed Venus Surface-Atmosphere Geochemistry Experiments (SAGE) mission concept, being considered under the NASA New Frontiers program for launch between 2007 and 2009, will include an orbiter and two or three landers that will study the Venusian atmosphere and surface geology. Figure 7.3 shows the descent and surface operations for one of the landers, based on Pioneer Venus' probes [35]. This mission has a very different profile from JIMO, due to the short spacecraft lifetime expected on the surface (about an hour). Component failures are expected to start occurring from the moment the landers enter the atmosphere. The Venusian environment is one of the most challenging addressed by NASA, with temperatures near the surface reaching 900F and the atmospheric pressure close to 1300psi. This timeline and the limited radio bandwith available from the surface for transmitting science data make reporting a fault and receiving recovery recommendations from ground control infeasible. To maximize the overall robustness of the probes, their computational hardware must be simple and robust, implying very limited resources.

SAGE is due to launch a few years from now. The travel time to Venus is several months, the exact duration depending on the route taken. Consequently, splitting development into two phases pre- and post-launch is probably neither practical nor useful. Similarly, it is unlikely that there will be a large payback to automating the trip to Venus.

From a spacecraft-level perspective, the crucial component of the mission is orbital insertion and the descent of the landers to the surface. In the case of Pioneer Venus, the probes generally survived most of the descent and one probe, the "day probe," survived impact, transmitting for another 68 minutes (which was longer than its descent). From the data returned, it appears that it was depletion of the batteries, rather than environmental factors that eventually caused the probe to fail [35] p. 103.

Two observations can be brought forward to future missions to Venus.

1. Only a single short window of opportunity for each probe to gather science is available, so a probe cannot afford to spend much if any time thinking about what it should be doing. Nor can it enter a "safe" mode and expect the mission team to fix it later.

2. One or more of the probes has a reasonable chance that it will survive longer than expected, possibly much longer. If this happens the probe needs to know what to do to take advantage of this opportunity.

Consequently the probe will need some form of capability for contingent execution, dependent on recognizing and responding to rapid, and radical, failures of the spacecraft's systems and instrumen-

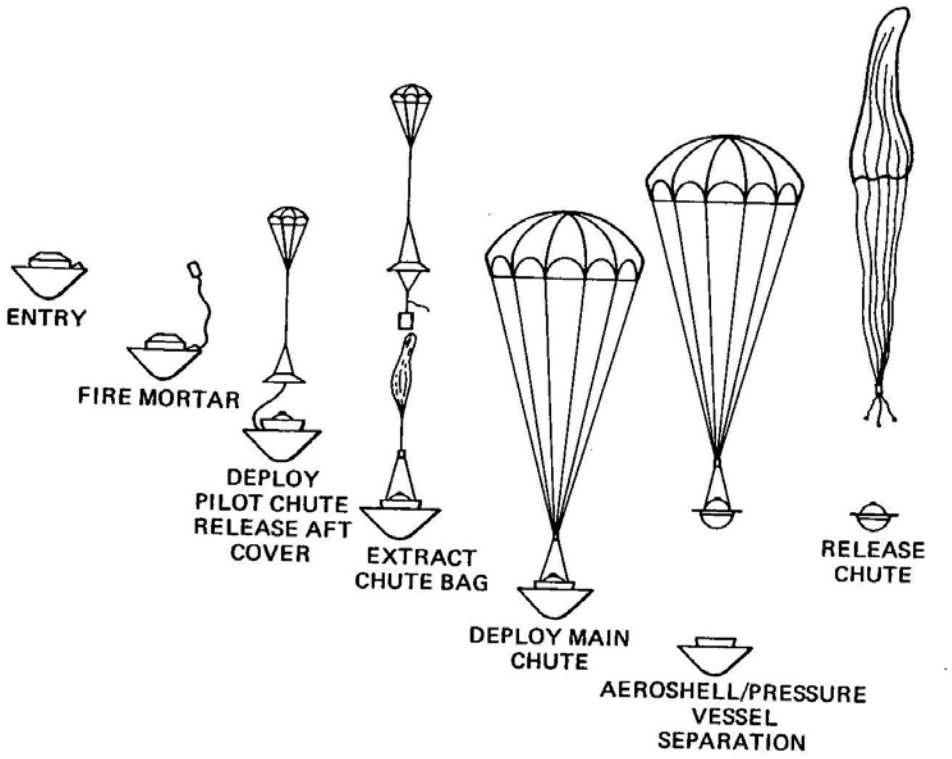

Figure 7.3: Venus Lander descent [35]

tation. Venus SAGE will be under very tight hard real-time constraints and will require an executive that operates deterministically and robustly within those constraints.

The orbiter's operations may also benefit from compiled approaches to automation. The orbiter will be in data capture mode for as long as the probes are actively transmitting with no possibility of resending missing data. How to respond to faults that occur during this interval is an issue that should be examined off-line, with the responses compiled, tested, and loaded pre-launch.

The spacecraft that comprise Venus SAGE will need to be extremely self-sufficient. Once the science mission begins, control from Earth will not be very useful. In fact, on Pioneer Venus, the probes were not designed to accept uplinked commands from Earth at all, once they were deployed. Similarly on future Venus probe missions, Earth-based personnel will lack the required situation awareness to properly react to observed events in time to help. Instead, the spacecraft will need to make all execution decisions itself based on goals received from Earth prior to the start of the mission. This includes contingencies as the spacecraft begins to fail. While the failure modes may be surmised prior to the descent, the exact order will be unknown and there will likely be some that were not foreseen by the mission designers.[2] Further, out of the several science instruments planned for each probe, exactly how long each one will last and the order in which they will fail cannot be predicted exactly. Here compiled techniques may play a role by providing a more inspectable artifact and, if designed properly, separating the execution engine from the knowledge base the engine uses. In this way, the engine can be implemented and validated separately from the knowledge base.

[2]Past mission descriptions frequently include the phrase "the mission team did not anticipate."

Chapter 8

Conclusions and Recommendations

In this section, we sum up the study, reiterate our conclusions, and propose some areas for further work. The overall conclusion we have arrived at on this study is that moving from an opportunistic and ad hoc approach (e.g., see Chapter 2) to compiled automation to an approach that is principled, predictable, and verifiable would be a huge win, whether or not the end result was to put the underlying reasoning processes on-board. The techniques required are known, the methods of implementation are consistent with current architectures, and their introduction can be gradual so as to reduce risk and ensure incremental rather than a radical transition in operational practices.

This report documents the results of an investigation into the methods and means for using compiled automation to speed the development and deployment of increasingly autonomous spacecraft. We have provided a precise definition of what we mean by compiled automation and delimited the scope of functions addressed in this study to be those that sit immediately above the current implementation of spacecraft executives as sequence execution engines. We presented a sampling of architectures within which it is proposed by various parties that these functions should be implemented and integrated, and surveyed a range of tactics for compiling inference to support those functions. Finally, we discussed the tradeoffs that must be considered in deciding whether and how to automate those functions, including compilation, and presented a pair of case studies, drawn from missions currently being studied or under development.

The approach we have taken in this report of considering the automation of individual functions, rather than "compiled autonomy" as a majestic and indivisible whole is driven from the underlying technology: it is difficult to talk about how to compile something without being fairly precise about what the "thing" is (in this case an algorithm, or a form of inference). This natural partitioning of functions is also very appropriate, however, considering the current status of spacecraft autonomy. A gradual increase in automation related to autonomy seems much more feasible (and more useful) than wholesale adoption of a completely autonomous system, both given issues of technology development and maturation, and the fact that autonomy is not properly all-or-nothing. Some level of human interaction will always be necessary with most missions, with the exact form and degree of that interaction depending on the nature of the mission. Further, that interaction will evolve over the life of the mission as the human mission team gains experience operating the spacecraft, the objectives evolve (e.g., opportunistic science or extended missions), or the spacecraft experiences unforeseen difficulties. Providing approaches such as compiled automation, that enable these operators to incrementally and confidently automate additional functions will provide benefits.

Compiled automation makes the following properties achievable for a wide range of styles of infer-

ence, thus supporting the use of that inference in on-board, resource-constrained, mission-critical functions:

Predictability – Inferential models and algorithms can be transformed in ways that permit rigorous guarantees on run-time, responsiveness, coverage (what inputs are accounted for), and correctness (getting the right output for a given input). As discussed in Chapter 5, this can be accomplished for search-based inference and other techniques for which such guarantees have historically been difficult to achieve.

Specialization – These functions can be adapted in context-specific ways, for example tailoring a function to a given domain, or compiling a rule-base to optimize a given class of queries.

Modularization – A compiled approach supports and encourages the use of architectures in which modules are individually generated and validated, then can be added to a larger system without the need for re-validating the overall system.

Smaller modules – One motivation for some forms of compilation is a reduction in size, for example to fit a model within strict memory limits. Moore's Law makes this a secondary benefit, but still in some circumstances potentially useful, for example, the effects of radiation will be harder to mitigate in smaller, more densely packed components.

Faster inference – Not infrequently, faster models are larger models, because the speedup is achieved through compilation in inference steps into the model itself. Again, while Moore's Law (and the lag in qualifying new hardware for space) means that flight hardware should continue a steady improvement in performance, there are and will remain applications where this issue is relevant (for example, on an atmospheric probe on Venus or one of the gas giants).

Our investigations have led us to some conclusions regarding the current state of the art in spacecraft autonomy, as well. First and foremost, substantial operational autonomy for complex spacecraft is well beyond the current state of the art. Hardware failures and other unexpected events can and will lead to significantly altered mission profiles. At the current state of the art, it is difficult to conceive of building an on-board, automated reasoner that could be counted on to respond appropriately to the failures dealt with during the Galileo mission, making the full scope of operational and configuration changes required. An incremental approach to increasing autonomy, rather than a "big-bang" move from the current system to full autonomy, is clearly indicated. As an additional architectural implication for spacecraft that are for reasons of distance or other communication barriers unable to phone home, diagnosis and recovery predicated on maintaining some form of homeostasis will in many cases be too limited. This has significant implications for the degree to which mission planning and execution must be integrated, or at the very least coordinated, with diagnosis and recovery.

To conclude this report, we identify several areas where these results could be extended and further work pursued.

- Rigorous Tradeoffs. Providing tools and techniques to add rigor and traceability to the design and implementation of automation (compiled or otherwise) for a given set of mission functions would be a very useful step. As design tradeoffs are made, capturing the design rationale in a form that permits rigorous evaluation of alternatives, and further supports reevaluation of design decisions over the life of the mission (e.g., as software is modified), would be a huge step forward.
- Impact on Hardware Architecture. A study is warranted to consider hardware solutions, such as FPGA (Field Programmable Gate Arrays), that can be reconfigured in orbit and have

high execution speeds. With a compiled approach, because the compiled artifacts are smaller and do not depend on a large, context-dependent model, a spacecraft can cannibalize itself when memory requirements run short and then quickly reinstate the components that were "swapped out" when space frees up. They can be restored either from Earth or from longer-term spacecraft memory.

- Assessment of Timing Constraints. It would be interesting to explore the use of compilation methods as a wedge to introduce a more generally rigorous treatment of timing constraints (on both inference and execution). Mission-level software is still too frequently described as "fast enough," rather than providing any kind of analysis of what the real constraints are, much less making tradeoffs against those constraints, at either design-time or run-time.
- Technique Selection. It would be good to have more experience with applying compilation methods to mission-level functions. On this project, a brief review of the current state of the art leads to two conclusions, first a lot of techniques can be applied to compile various forms of inference, and second which technique is best and how it should be applied in a given situation is highly context-dependent, in ways that are not yet understood.

Any of these next steps could be pursued independently in a broad sense, by looking at several different mission types, or could be addressed within the context of a specific mission. One interesting avenue would be to develop a compiled automation "strategy" for a specific future mission along the lines of the JIMO and SAGE sketches provided in Chapter 7 that would address each of the issues identified above. Assuming the mission is defined in sufficient detail (e.g., mission objectives, basic spacecraft functionality, etc.), this approach should generate an executable strategy for realizing the benefits of compiled automation detailed in this report.

Bibliography

[1] P. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp. 268–272, Seattle, WA., 1987.

[2] AIAA Standards Committee, editor, *AIAA Guide for the Verification and Validation of Computational Fluid Dynamics Simuations (G-077-1998)*, AIAA, 1998.

[3] S. Y. Alan. *Inductive Policy Selection for First-Order MDPs.*

[4] R. Aler, D. Borrajo, and P. Isasi, "Knowledge Representation Issues in Control Knowledge Learning," in *Proc. 17th International Conf. on Machine Learning*, pp. 1–8. Morgan Kaufmann, San Francisco, CA, 2000.

[5] D. Bernard, G. Dorais, C. Fry, E. Jr, B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. Nayak, B. Pell, K. Rajan, N. Rouquette, B. Smith, and B. Williams, "Design of the remote agent experiment for spacecraft autonomy," in *Proceedings of the IEEE Aerospace Conference*, 1998.

[6] D. S. Bernstein and S. Zilberstein, "Reinforcement Learning for Weakly-Coupled MDPs and an Application to Planetary Rover Control," in *Proceedings of the 6th European Conference on Planning (ECP)*, Toledo, Spain, September 2001.

[7] D. S. Bernstein, S. Zilberstein, R. Washington, and J. L. Bresina, "Planetary Rover Control as a Markov Decision Process," in *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada, June 2001.

[8] A. Blum and M. Furst, "Fast Planning Through Planning Graph Analysis," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pp. 1636–1642, 1995.

[9] M. Boddy and R. Goldman, "Empirical Results on Scheduling and Dynamic Backtracking," in *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation for Space*, 1994.

[10] B. Bonet and H. Geffner, "Planning with Incomplete Information as Heuristic Search in Belief Space," in *Artificial Intelligence Planning Systems*, pp. 52–61, 2000.

[11] B. Bonet and H. Geffner, "GPT: A Tool for Planning with Uncertainty and Partial Information," in *Workshop on Planning with Uncertainty and Partial Information, International Joint Conf. of AI*, pp. 82–87, Seattle, WA., 2001.

[12] G. Boothroyd, P. Dewhurst, and W. Knight, *Product Design for Manufacture and Assembly*, Marcel Dekker, 1994. ASIN 0824791762.

[13] C. Boutilier, T. Dean, and S. Hanks, "Decision-Theoretic Planning: Structural Assumptions and Computational Leverage," *Journal of Artificial Intelligence Research*, vol. 11, pp. 1–94, 1999.

[14] G. E. Box, W. G. Hunter, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenter: An Introduction To Design, Data Analysis, And Model Building*, John Wiley and Sons, New York, 1978. ISBN 0-47109-31-57.

[15] J. R. Boyd. *The Essence of Winning and Losing.* http://www.d-n-i.net/second_level/boyd_military.htm, 1996.

[16] R. A. Brooks, "Intelligence Without Representation," *Artificial Intelligence*, vol. 47, pp. 139–159, 1991.

[17] J. Casani, "Mission Characteristics Overview to the Forum on Concepts and Approaches for Jupiter Icy Moons Orbiter," in *Forum on Concepts and Approaches for Jupiter Icy Moons Orbiter, http://www.lpi.usra.edu/meetings/jimo2003/*, Clear Lake, TX, June 2003.

[18] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling," in *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS2000)*, Breckenridge, CO, April 2000.

[19] S. Chien, R. Sherwood, G. Rabideau, R. Castano, A. Davies, M. Burl, R. Knight, T. Stough, J. Roden, P. Zetocha, R. Wainwright, P. Klupar, J. Van Gaasbeck, P. Cappelaere, and D. Oswald, "The Techsat-21 autonomous space science agent," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pp. 570–577. ACM Press, 2002.

[20] S. Chowdhury, *Design for Six Sigma: the revolutionary process for achieving extraordinary profits*, Dearborn Trade Publishing, Chicago, IL, 2002. ISBN 0-7931-5224-0.

[21] A. Darwiche. *New Advances in Structure-Based Diagnosis: A Method for Compiling Devices*, 1997.

[22] A. Darwiche and P. Marquis, "A Knowledge Compilation Map," *Journal of Artificial Intelligence Research*, vol. 17, pp. 229–264, 2002.

[23] A. Darwiche and G. M. Provan, "Query DAGs: A Practical Paradigm for Implementing Belief-Network Inference," *Journal of Artificial Intelligence Research*, vol. 6, pp. 147–176, 1997.

[24] R. Dearden and D. Clancy, "Particle Filters for Real-time Fault Detection in Planetary Rovers," in *International Workshop on Diagnosis (DX)*, 2001.

[25] R. Dearden, N. Meuleau, S. Ramakrishnan, D. Smith, and R. Washington, "Contingency Planning for Planetary Rovers," in *NASA 2002 Planning and Scheduling Workshop*, 2002.

[26] E. M. DeJong, S. R. Levoe, J. M. McAuley, B. A. McGuffie, and P. M. Andres, "Automating Planetary Mission Operations," in *Reducing the Cost of Spacecraft Ground Systems and Operations (RCSGSO 2003)*, Pasadena, CA, July 2003.

[27] M. B. Dias, S. Lemai, and N. Muscettola, "A Real-Time Rover Executive Based On Model-Based Reactive Planning," in *The 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, May 2003.

[28] M. Drummond and J. Bresina, "Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction,", pp. 138–144, 1990.

[29] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks, "Software Architecture Themes in JPL's Mission Data System," in *AIAA Space Technology Conference and Exposition (AIAA-99)*, Albuquerque, NM, September 1999.

[30] S. Edelkamp and M. Helmert, "Exhibiting Knowledge in Planning Problems to Minimize State Encoding Length," in *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pp. 135–147, New York, 1999, Springer-Verlag.

[31] A. Elson, "Software Lifecycle Themes for JPL Mission Data System (MDS) Project," in *AIAA Space Technology Conference and Exposition (AIAA-99)*, Albuquerque, NM, September 1999.

[32] T. Estlin, R. Castano, B. Anderson, D. Gaines, F. Fisher, and M. Judd, "Learning and Planning for Mars Rover Science," in *IJCAI 2003 workshop notes on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World modeling, planning, learning, and communicating*, Acapulco, Mexico, August 2003.

[33] T. Estlin, F. Fisher, D. Gaines, C. Chouinard, S. Schaffer, and I. Nesnas, "Continuous Planning and Execution for an Autonomous Rover," in *Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space*, Houston, TX, October 2002.

[34] T. Estlin and R. Mooney, "Learning to Improve Both Efficiency and Quality for Planning," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, (IJCAI-97)*, Nagoya, Japan, August 1997.

[35] R. O. Fimmel, L. Colin, and E. Burgess, *Pioneer Venus (NASA SP-461)*, NASA Scientific and Technical Information Branch, 1983.

[36] R. J. Firby, "Adaptive Execution in Complex Dynamic Worlds," Technical Report YALEU/CSD/RR672, Yale University, 1989.

[37] M. Fox and D. Long, "Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub- problems in Planning," in *IJCAI*, pp. 445–452, 2001.

[38] E. Gat, "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents," in *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, July 1992.

[39] M. L. Ginsberg, "Universal Planning: An (Almost) Universally Bad Idea," *AI Magazine*, vol. 10, no. 4, pp. 40–44, 1989.

[40] R. P. Goldman, D. J. Musliner, K. D. Krebsbach, and M. S. Boddy, "Dynamic Abstraction Planning," in *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pp. 680–686, Providence, Rhode Island, 1997, AAAI Press / MIT Press.

[41] R. P. Goldman, D. J. Musliner, and M. J. Pelican, "Using Model Checking to Plan Hard Real-Time Controllers," in *Proceeding of the AIPS2k Workshop on Model-Theoretic Approaches to Planning*, Breckeridge, Colorado, April 2000.

[42] M. Goldszmidt and J. Pearl, "Qualitative probabilities for default reasoning, belief revision and causal modeling," *Artificial Intelligence*, vol. 84, no. 1–2, pp. 57–112, 1996.

[43] J. J. Grefenstette, C. L. Ramsey, and A. C. Schultz, "Learning Sequential Decision Rules Using Simulation Models and Competition," *Machine Learning*, vol. 5, pp. 355–381, 1990.

[44] V. A. Ha and D. J. Musliner, "Balancing Safety Against Performance: Tradeoffs in Internet Security," in *Proc. Hawaii Int'l Conf. on System Sciences*, January 2003.

[45] D. M. Harland, *Jupiter Odyssey*, Springer-Praxis, 2000. About the Galileo mission.

[46] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1999.

[47] Interface and I. Control Systems. *What Is SCL*.

[48] A. K. J'onsson and J. Frank. *A framework for dynamic constraint reasoning using procedural constraints*, 2000.

[49] A. K. Jonsson, P. H. Morris, N. Muscettola, K. Rajan, and B. D. Smith, "Planning in Interplanetary Space: Theory and Practice," in *Artificial Intelligence Planning Systems*, pp. 177–186, 2000.

[50] R. Khardon, "Learning Action Strategies for Planning Domains," *Artificial Intelligence*, vol. 113, no. 1-2, pp. 125–148, 1999.

[51] P. Kim, B. C. Williams, and M. Abramson, "Executing Reactive, Model-based Programs through Graph-based Temporal Planning," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 2001. Discusses RMPL and Kirk, which does graph search on TPNs.

[52] M. G. Lagoudakis and R. Parr. *Value Function Approximation in Zero-Sum Markov Games*.

[53] M. Martín and H. Geffner, "Learning Generalized Policies in Planning Using Concept Languages," in *Proceedings of the 7th International Conference on Knowledge Representation and Reasoning*. Morgan Kaufmann, April 2000.

[54] N. Meuleau and D. E. Smith, "Optimal Limited Contingency Planning," in *ICAPS-03 Workshoop on Planning Under Uncertainty*, 2003.

[55] A. L. L. Michael P. Georgeff, "Reactive Reasoning and Planning," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp. 677–682, Seattle, WA, 1987. PRS.

[56] D. C. Montgomery, *Design And Analysis Of Experiments*, John Wiley and Sons, 5th edition, 2000. ISBN 0-47131-64-90.

[57] D. E. Moriarty, A. C.Schultz, and J. J. Grefenstette, "Evolutionary Algorithms for Reinforcement Learning," *Journal of Artificial Intelligence Research*, vol. 11, pp. 199–229, 1999.

[58] N. Muscettola. *Remote Agent*. http://ic.arc.nasa.gov/projects/remote-agent, 2000.

[59] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. *IDEA: Planning at the Core of Autonomous Reactive Agents*, 2001. Seems to have been at some AAAI event but unknown which.

[60] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams, "Remote Agent: To Boldly Go Where No AI System Has Gone Before," *Artificial Intelligence*, vol. 103, no. 1-2, pp. 5–47, 1998.

[61] D. Musliner and R. Goldman. *CIRCA and the Cassini Saturn orbit insertion: Solving a prepositioning problem*, October 1997.

[62] D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: A Cooperative Intelligent Real-Time Control Architecture," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 6, pp. 1561–1574, 1993.

[63] D. J. Musliner, E. H. Durfee, and K. G. Shin, "World Modeling for the Dynamic Construction of Real-Time Control Plans," *Artificial Intelligence*, vol. 74, no. 1, pp. 83–127, March 1995.

[64] D. J. Musliner, M. J. Pelican, and K. D. Krebsbach, "Building Coordinated Real-Time Control Plans," in *Proc. Third Annual International NASA Workshop on Planning and Scheduling for Space*, October 2002.

[65] NASA. *Mars Climate Orbiter from NSSDC Master Catalog.* "http://nssdc.gsfc.nasa.gov/database/MasterCatalog?sc=1998-073A", September 2000.

[66] NASA. *The End of an Asteroidal Adventure Shoemaker Phones Home for the Last Time.* http://nssdc.gsfc.nasa.gov/planetary/text/near_pr_20010228.txt, February 2001.

[67] NASA JPL. *Jupiter Icy Moons Orbiter Home Page.* http://www.jpl.nasa.gov/jimo/.

[68] P. P. Nayak and B. C. Williams, "Fast Context Switching in Real-Time Propositional Reasoning," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence (IJCAI-97) and the Ninth Innovative Applications of Artificial Intelligence Conference*, T. Senator and B. Buchanan, editors, pp. 50–56, Menlo Park, California, 1998, AAAI Press.

[69] L. Ngo, P. Haddawy, and J. Krieger. *Efficient Temporal Probabilistic Reasoning Via Context-Sensitive Model Construction*, 1997.

[70] A. P. Nikora, N. F. Schneidewind, and J. C. Munson, "Practical Issues In Estimating Fault Content And Location In Software Systems," in *AIAA Space Technology Conference and Exposition (AIAA-99)*, Albuquerque, NM, September 1999.

[71] R. Patrick and F. Croce, "Autonomous Operations Through Procedure Automation," in *Proceedings of Reducing the Cost of Spacecraft Ground Systems and Operations 2003*, Pasadena, CA, July 2003.

[72] R. W. Proud, J. J. Hart, and R. B. Mrozinski, "Methods for Determining the Level of Autonomy to Design into a Human Spaceflight Vehicle: A Function Specific Approach," in *To be published*, NASA Johnson Space Center, 2003.

[73] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, and A. Govindjee, "Iterative Repair Planning for Spacecraft Operations in the ASPEN System," in *International Symposium on Artificial Intelligence Robotics and Automation in Space*, Noordwijk, Netherlands, June 1999.

[74] G. Rinker, "Mission Data System Architecture and Implementation Guidelines," in *Ground System Architectures Workshop*, El Segundo, California, March 2002.

[75] R. L. Rivest, "Learning Decision Lists," *Machine Learning*, vol. 2, no. 3, pp. 229–246, 1987.

[76] S. Rosenschein, "Formal Theories of Knowledge in AI and Robotics," Technical Report CSLI-87-84, Center for the Study of Language and Information, 1987.

[77] B. Selman and H. Kautz, "Knowledge compilation and theory approximation," *Journal of the ACM*, vol. 43, no. 2, pp. 193–224, 1996.

[78] R. Sherwood, S. Chien, D. Tran, B. Cichy, R. Castano, A. Davies, and G. Rabideau, "Next Generation Autonomous Operations on a Current Generation Satellite," in *Proceedings of Reducing the Cost of Spacecraft Ground Systems and Operations 2003*, Pasadena, CA, July 2003.

[79] Unknown. *Polar Orbiting Satellites.* http://www.ice.mtu.edu/online_docs/TeraScan-3.2/html/home_basic/polar_sats_overview.htm, September 2002.

[80] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe, "Integrating planning and learning," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 7, no. 1, pp. 81–120, 1995.

[81] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARaTY Architecture for Robotic Autonomy," in *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky, Montana, March 2001.

[82] M. Webb. *Fobos 1F*. http://www.astronautix.com/craft/fobos1f.htm, June 2002.

[83] J. R. Wertz, "Autonomous Navigation and Autonomous Orbit Control in Planetary Orbits as a Means of Reducing Operations Cost," in *Proceedings of Reducing the Cost of Spacecraft Ground Systems and Operations 2003*, Pasadena, CA, July 2003.

[84] B. C. Williams, M. Ingham, S. H. Chung, and P. H. Elliott., "Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers," *Proceedings of the IEEE*, vol. 9, no. 1, pp. 212–237, January 2003. Special Issue on Modeling and Design of Embedded Software.

[85] B. C. Williams and P. P. Nayak., "A Reactive Planner for a Model-based Executive," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 1997.

[86] S. Zilberstein, R. Washington, D. S. Bernstein, and A.-I. Mouaddib, "Decision-Theoretic Control of Planetary Rovers," in *Proceedings of the Dagstuhl Workshop on Plan-based Control of Robotic Agents*, Wadern, Germany, October 2001.