

A Hybrid Procedural/Deductive Executive For Autonomous Spacecraft ^{*}

Barney Pell [†] Ed Gamble [§] Erann Gat [§] Ron Keesing [†] Jim Kurien [†]
Bill Millar [†] P. Pandurang Nayak [‡] Christian Plaunt [†] Brian Williams [‡]

Abstract

The New Millennium Remote Agent (NMRA) will be the first AI system to control an actual spacecraft. The spacecraft domain places a strong premium on autonomy and requires dynamic recoveries and robust concurrent execution, all in the presence of tight real-time deadlines, changing goals, scarce resource constraints, and a wide variety of possible failures. To achieve this level of execution robustness, we have integrated a procedural executive based on generic procedures with a deductive model-based reasoning system. The resulting system enables designers to code knowledge via a combination of procedures and declarative models, yielding a rich modeling capability suitable to the challenges of real spacecraft control.

1 Introduction

We are developing the first on-board AI system to control an actual spacecraft. The mission, Deep Space One (DS-1), is the first in NASA's New Millennium Program (NMP), an aggressive series of technology demonstrations intended to push Space Exploration into the 21st century. DS-1 will launch in mid-1998 and will navigate and fly by asteroids and comets, taking pictures and sending back information to scientists on Earth. One key technology to be demonstrated is spacecraft autonomy, including robust plan execution. Since planning on a spacecraft is a precious activity (Pell *et al.* 1997b), execution of plans must be highly robust. Hence,

This paper appears in the working notes of the AAI Fall Symposium on Model-Directed Autonomous Systems

[‡]Recom Technologies, NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.

[†]Caelum Research, NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.

[§]Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109.

the execution system must maintain its safety and successfully execute the plan, even in the presence of hardware faults and other unexpected events.

This work is being implemented as part of the New Millennium Remote Agent (NMRA) architecture (Pell *et al.* 1997a). This architecture integrates traditional real-time monitoring and control with constraint-based planning and scheduling (Mussettola 1994), robust multi-threaded execution (Gat 1996), and model-based diagnosis and reconfiguration (Williams & Nayak 1996).

Pell *et al.* (1997b) describes the approach we have taken to the automatic generation of robust plans, which leave flexibility to be used by the execution system in case problems arise during execution. This paper focuses on the execution system itself. In particular, we found it necessary to develop a hybrid procedural and deductive executive in order to achieve the high levels of reliability required in the autonomous spacecraft domain. This paper discusses our domain, the component execution technologies, and the approach we took to integrating these technologies into a hybrid executive which was better than either approach alone.

The paper is organized as follows. Section 2 discusses the spacecraft domain and requirements which influence our design. Section 3 describes our approach to integrating procedural and deductive execution systems. Section 4 describes the capabilities in our procedural executive. Section 5 addresses the capabilities in the Livingstone model-based execution system. Section 6 shows how we have integrated the two systems. Section 7 discusses some key points about our design. We then consider related work and conclude.

2 Domain and Requirements

The autonomous spacecraft domain presents a number of challenges for robust plan execution.

Interacting Concurrent Processes

Many devices and systems must be controlled, leading to multiple threads of complex activity. These concurrent processes must be coordinated to control for negative interactions, such as vibrations of the thruster system violating stability requirements of the camera. Also, activities may have precise real-time constraints, such as taking a picture of an asteroid during a narrow window of observability.

High Reliability

Moreover, a central requirement of spacecraft operation is *high reliability*. Since a spacecraft is very expensive and often unique, it is essential that it achieve its mission with a very high level of reliability. Part of this high reliability is achieved through the use of very reliable hardware. However, the harsh environment of space or the inability to test in all flight conditions can still cause unexpected hardware failures, so that the software architecture is required to compensate for such contingencies. This requirement dictates the use of an executive and elaborate system-level fault protection that can rapidly react to contingencies by retrying failed actions, reconfiguring spacecraft subsystems, or safing the spacecraft to prevent further, potentially irretrievable, damage.

Interacting Recoveries

A particularly challenging problem in the design of a spacecraft fault protection system arises from the combination of the above two properties: recovering failed activities in the presence of concurrent activity. As an example, consider two spacecraft subsystems in DS-1: the engine gimbal (EG) and the solar panel gimbal (SPG). A gimbal is part of a system that enables it to rotate. For example, the engine nozzle can be rotated to point in various directions without changing the spacecraft orientation, and the solar panels can be independently rotated to track the sun. In DS-1, both sets of gimbals communicate with the main computer via a shared board called the gimbal drive electronics (GDE). If either system experiences a communications failure, one way to reset the system is to power-cycle (turn on and off) the GDE. However, resetting the GDE to fix one system also resets the communication to the other other system. In particular, resetting the engine gimbal, to fix an engine problem, causes us to temporarily lose control of the solar panels. Thus fixing one problem can cause new problems to arise. To avoid this, the recovering system needs to take account of global constraints rather than just mak-

ing local fixes in an incremental fashion. Examples like this drove the design of our hybrid execution system.

3 Approach

In this section we first describe the problem we faced, and then our approach to solving it.

The Problem

Complex execution of spacecraft plans requires capabilities of both procedural and declarative execution systems.

On the one hand, execution needs reactivity, time-sensitivity, and sophisticated control constructs. Such constructs include loops, parallel activity, locks, and synchronization. The standard approach to this is to build executives which interpret directives in a rich procedural language, make fast choices based on contextual knowledge, and choose alternatives when previous choices fail.

However, this strict procedural approach has its limitations. Execution also requires the management of configurations of components with different capabilities and costs. Robust recovery may require novel combinations of actions and some amount of projection in order to trade off costs and benefits. For example, the propulsion system on the Cassini spacecraft (Brown, Bernard, & Rasmussen 1995) has a very complex set of valves. These include explosive *pyro* valves which can change states only once, and also valves with varying amounts of wear and tear. It is very difficult to express in a procedural way the right valve choices to redirect fluid flow while minimizing costs and risks.

On the other hand, a deductive executive of the form developed by Williams & Nayak (1996) can solve such problems quickly enough to be useful during on-line execution. Yet, the models which can be reasoned about by such systems lack the flexibility and richness of activity description found in procedural execution systems. For example, the Livingstone system (Williams & Nayak 1996) is based on a propositional logic which does not model time, loops, or synchronization. Thus it is hard to encode knowledge like the following:

To send a signal down to earth via an antenna, first turn off the antenna's exciter, then turn on the antenna's power supply, wait 5 seconds, and turn the exciter on again.¹

¹The reason for this requirement is that turning on the power supply sends a surge of power which would destroy the sensitive exciter. Hence the exciter should

Hybrid Approach

From this we see that both approaches to execution have strengths and weakness, which are complementary. Our approach is to develop a hybrid executive, as follows:

- Use a procedural executive for time, control knowledge, schedule execution, hierarchical task decomposition, and routine configuration handling.
- Use a deductive executive for state inference, novel responses based on global context, and cost-benefits analysis.
- Work out clear interfaces between to the two systems to exploit the strengths of each.

Note that some divisions are arbitrary, since certain capabilities exist in both systems. This gives the designer flexibility to choose the best system and language for specific purposes.

In the next sections we describe the procedural executive and the model-based deductive executive. To reflect their roles in the NMRA, we will often refer to the procedural executive as *Exec* and the deductive executive as *MIR*, the mode-identification and reconfiguration system.

4 Procedural Executive

Our procedural executive is based upon a sophisticated scripting language (ESL) (Gat 1996) for describing control constructs necessary for execution. Such constructs manage concepts of time, events, multiple methods, class hierarchies, and generic procedures. An executive also needs a source of source of state update knowledge. In NMRA, the executive benefits from being abstracted from the hardware details by relying on the results of the mode identification (MI) component of Livingstone.

We briefly list some of the key constructs we have developed within the procedural executive:

Maintained Properties

`(with-maintained-properties <properties> body)`

- If properties are all currently true, the body is executed.
- If properties are false, the executive tries to achieve them first.

be switched off while the surge is happening, and then switched on again.

- Once they are true, the executive locks the properties and executes body.
- If the properties become false during execution of body, signal this loss and let context of body choose response.

Achieving properties

`(achieve <property>)`

- If this is the first thread to request the property, then execute an achievement method for the property.
- When achievement is successful, signal other waiting threads.
- If some other thread is already achieving the property, then wait for it to finish.
- If property is inconsistent with a current lock, either wait for lock to be released or fail immediately (based on preferences set by the invoking thread).

Device Management Idioms

Devices and classes are formalized using generic descriptions. Individual devices, switches, etc. are then modeled as instances of these classes.

```
(define-device-class :camera
  :power-function #'fsc-power-request
  :talk-function #'camera-talk-msg)

(define-device :camera_A :camera
  :health-state :ok
  :power-state :on
  :powered-thru :power-bus-1
  :switched-thru :fsc_camera_sw1
  :ready-state ((:health_state :ok)
                (:power_state :on))
  :selected-device t)
```

Based on these device idioms, we have defined generic procedures for device configuration and management:

```
(with-selected-device <class>
  (do-activity))
```

This construct selects a device of the class, achieves its ready-state, and then locks the properties of that ready-state and maintains them as it executes the enclosed activity.

Recovering failed properties

In the case where a maintained property is lost (for example, a device makes an uncommanded and erroneous state transition), the enclosing context of the (`with-maintained-properties`) form determines the appropriate response.

```
(with-automatic-recoveries
  body)
```

This form indicates that the response to lost properties within *body* is to suspend the thread while waiting for an automatic recovery, and then retry the body.

On the other hand, if no enclosing context handles the lost properties notification, then (`with-maintained-props body`) fails.

Automatic Recoveries Thread A special thread in the executive manages the property locks. Whenever some property lock is violated:

1. Suspend all tasks who have a violated lock.
2. Ask for an automatic recovery for all required locks.
3. Wait for a recovery action to be generated in response to this query.
4. Execute the recovery action.
5. Signal `recovery-event`.

The effective of signaling `recovery-event` is to wake up all threads who were suspended waiting for a recovery. Each awakened thread then retries the body, attempting to re-establish the required properties.

For properties which were restored by the recovery action, this will succeed. For properties which are still failed, the affected threads will block again, and wait for another recovery step.

At any time when the `automatic-recoveries` thread fails to return with a recovery action while some threads are blocking on required properties, the waiting tasks fail automatically. This can happen either when the recovery expert believes no further actions need be achieved, or when it fails to find a solution to the recovery request.

5 Deductive Executive

In the NMRA architecture, Livingstone plays the role of a Mode Identification and Reconfiguration system (MIR). The Livingstone system has been well described elsewhere (Williams & Nayak 1996;

1997), so we refer only to the critical properties here.

The most interesting property in terms of execution capability is the Mode Reconfiguration (MR) component of Livingstone. MR performs a function similar to STRIPS-style planning (Fikes, Hart, & Nilsson 1972). Namely:

- **Given:**
 - a *target state*, which specifies a desired mode for each of a set of components, and
 - a probability distributions on the current state, as generated by Livingstone’s Mode Identification (MI) component;
- **Generate:** a sequence of actions to move from the current state to the target state.

The sequence can be null, meaning that no action is necessary, or the process can fail, meaning that no such plan could be found. Each step in the sequence is a “primitive” operator from the perspective of Livingstone’s models. If Livingstone functions as a stand-alone configuration management system (Williams & Nayak 1996), each primitive would correspond to a command directly executable by the lower-level flight software.

6 Integration

Having described the Procedural Executive and the Deductive Executive, we now describe how we combine these systems to form an integrated Hybrid Executive. The integration effort required:

- An interface between the Procedural and Deductive Executives
- Constraints on the Procedural Executive
- Constraints on the Deductive Executive

Exec/MIR Interface

Given descriptions of the two components, the basic form of the interface is relatively clear.

Exec asks MIR to recover the system to a target state as specified by a set of constraints. Some of these constraints are true at the time, and some are not. Requested constraints may be false in the current state either because they are being requested to be made true via this mechanism, or because they were true and have now been invalidated and we want them recovered now.

Given the request, and MIR’s knowledge of the current state, MIR generates a response telling EXEC to do a recovery step. There are two points

to note here. First, what appears to MIR to be a “primitive” action can in fact be a complex procedure executable by the EXEC. This gives a considerable level of robustness to the individual operations in a recovery plan. Second, our interface has MIR send one step at a time, even though it generated internally an entire sequence to achieve the goals. This decision adds a considerable reactivity to the system, as between steps MIR is free to take account of new actions and generate a new plan at the next request. It also reduces the complexity of MIR’s models, which simplifies the recovery planning process.

Exec then executes the action, and if it needs more recovery steps it forms another query, and so on until either all constraints are satisfied or Exec aborts the unsatisfied goals.

Constraints on the Procedural Executive

In our integration, the Procedural Executive is the one who requests recovery assistance from the deductive system, and also the one who acts on the results. The key issues from Exec’s view are as follows:

- when does Exec recognize the need to recover?
- what does Exec do before asking for recovery (to stabilize)?
- what does Exec ask for?
- how does an executed recovery interact with the rest of the execution context?
- when does the process terminate?

Our answers to these questions are relatively transparent given the description of the Procedural Executive in Section 4.

The executive code is designed in such a way that each procedure asserts the properties that it requires maintained over its interval of execution, using the (`with-maintained-properties`) construct. In this way, the Exec understands the constraints which support the entire current execution context. In the event that some properties are lost or otherwise unachievable without the help of the deductive system, Exec suspends the unsupported threads. Then it formulates a query based on both the active and desired constraints (via the `automatic-recoveries` thread), and sends the query off to MIR.

When MIR returns an action, the action is executed, and then the unsupported threads are re-activated to check if their conditions are true. Note that most Exec procedures count the number of times they have retried a particular approach, and try something else or give up if this retry counter gets too high.

The `automatic-recoveries` thread remains in action forever, so unsatisfied constraints following execution of some recovery step will lead to a new recovery request.

In terms of termination of this process, a particular round of recovery generation (marked by a fixed set of constraints active in the Exec) ends when one of the following is true:

- When all the constraints are satisfied.
- When some thread has relaxed a constraint, e.g., by aborting or switching to another method to achieve its purpose. In this case, there may still be need for recoveries, but the request will be different due to relaxed constraints. This case also covers the situation in which many threads are aborted due to a violation of a higher-level temporal constraint in the plan being executed or another form of plan failure (Pell *et al.* 1997b).
- When the executive detects a simple loop in the recovery dialogue.²

Constraints on the Deductive Executive

When used as a stand-alone configuration system, Livingstone is free to generate any sequence of steps so long as it achieves the goals in the end. However, as part of an integrated executive, there are constraints on possible actions MIR can generate. Most significant is the following:

CONSTRAINT: MIR will not suggest a step which would falsify those required constraints which are currently true.

Restated, this constraint means that MIR won’t generate a recovery step that would make the situation worse, relative to the posted constraints. Combined with the requirement on Exec to constrain all required properties as part of a recovery request, this restriction on MIR prevents the problems to do with resource preemption and interacting recoveries discussed in Section 2. Without this constraint,

²We use a simple form of loop detection in which a loop is indicated by a high value of a counter, which is incremented each time a recovery is requested, and which is reset each time a constraint is relaxed.

some perfectly healthy activity that didn't fail (for example, the solar panel control loop) could be destroyed in an attempt to fix another component (for example, the engine gimbal controller).

7 Discussion and Future Work

Compositionality and Modularity

A major design goal within the NMRA is to develop modular and compositional representations of subsystems within the spacecraft domain. By modular, we mean that changes in the details of one subsystem should not force us to rewrite the code for most other systems. By compositional, we mean that we should be able to design and test code and models for independent subsystems, and then just run both sets of code together without experiencing harmful interactions.

For example, consider a camera system and an antenna system on a spacecraft. Suppose the camera system contains only one camera, which can jam occasionally, and that a simple reset command will fix it. Suppose the antenna system contains two alternate antennas, either one of which would suffice for communication to ground, and either one of which might fail permanently.

We define procedures in Exec to use the camera for taking pictures, which state that a picture-taking activity requires the camera to be on and healthy. We also have procedures to use the antenna system for down-linking data to earth. The procedures say the executive should first try to turn on antenna-1, and if that doesn't work, then it should turn antenna-1 off, wait 5 seconds, and then try to turn on antenna-2. In any case, the down-link activity requires that the antenna being used is healthy.

When we test these procedures separately, the recovery interface works just fine. If the camera breaks, Exec is informed that the required health property has been lost, and formulates a recovery request which constrains the target state to have the camera healthy, along with all other properties required by currently executing (and successful) procedures for other subsystems. MIR knows about the fix to the camera, and recommends this, which solves the problem. Similarly, if the current antenna breaks during down-link, Exec asks MIR to recover the health of antenna-1, in conjunction with all other currently satisfied properties required by other procedures. MIR knows that there is no recovery for the antenna, so it sends back a failed recovery message. This causes Exec's down-link procedure to fail, and then try an alternative method:

turn off antenna-1, wait 5 seconds, and then turn on antenna-2. This solves the problem.

Now consider the case where we executed the picture taking activity *and* the down-link activity concurrently. If only one of these systems breaks, the behavior is as described above: the problem is fixed. However, consider the case where *both* systems break. Since these are totally independent systems, it seems like the recoveries should not interact. However, the scenario with the current recovery protocol proceeds as follows. The Exec recognizes that both the camera and the antenna-1 have failed, and includes their health constraints as part of its recovery request to MIR. MIR tries to find a plan to repair *both* the camera and antenna-1. Since there is no repair for antenna-1 (as described in the case where this is the only failure, above), there is no repair for the conjunction of failures. Hence, MIR fails to generate a recovery. In this case, the picture activity and the down-link activity both fail. While the down-link activity still has another option, which it tries at this point, the picture activity (as described above) has no other options, and would be forced to abort. Whereas the individual failure cases were both handled perfectly, the combined failure was not handled, even though the affected systems are entirely independent.

We have found three responses to this problem.

First, note that this problem only happened in the case of simultaneous double independent failures. Such events are unlikely, as it multiplies the probability of each independent failure. Hence, most spacecraft fault protection systems do not design to double-fault robustness. In our case, the resulting plan failure will cause all the activities to abort and relinquish their constraints. After this, the Exec incrementally regains critical activities. Because it focuses on one subsystem at a time, MIR will be able to solve each posed problem. Hence the only real loss is one of plan execution robustness in the presence of multiple faults, which is desirable but not required.

Second, the above double-fault scenario failed because the picture-taking activity gave up the first time the recovery system failed to find a recovery. Most threads in the Exec are robust, and try the same activity multiple times if there are no alternatives. In this case, after the first "no recovery available" failure, the picture activity would wait a few seconds and then try again to request the healthy camera. In the meantime, the down-link activity would switch from antenna-1 to antenna-2, relaxing the unsatisfiable constraint on the perma-

nently failed antenna-1. Hence, the next recovery invocation would just take place with the health of the camera being the only unsupported constraint, and this single-fault case works fine as in the single-fault example above. So in general, some over-constrained multiple-failure cases are solved just by persistent attempts on the part of Exec. However, this is not a totally satisfying solution, because we can imagine multiple failures with multi-stage recoveries, for which it is unlikely that a few retries per activity will result in eventual success.

A third solution, which remains to be explored in future work, is to revise the recovery protocol to accept partial solutions rather than total solutions. The current planning problem for MIR is as follows:

Total Recoveries: MIR only generates a recovery to the executive if MIR finds a sequence of recovery steps (a recovery plan) that would restore *all* of the constrained conditions.

Instead, we could task MIR with the following planning problem:

Partial Recoveries: MIR only generates a recovery to the executive if MIR finds a sequence of recovery steps (a recovery plan) that would restore *at least one* of the constrained conditions that was not true in the current state, while maintaining all the constrained conditions that are true in the current state.

This revised interface constraint yields a compositional solution in our example. The picture and down-link activities both fail, and Exec asks MIR to recover with the entire set of constraints. MIR finds a plan which maintains all the currently true conditions and which recovers the camera (one of the currently unsatisfied constraints), even though this plan doesn't satisfy all the constraints as it leaves the antenna broken. Exec performs the suggested fix, and the picture activity is restored and resumes. At this point, the antenna still was not fixed, so Exec tries another recovery request, which fails as in the original example, and Exec now switches antennas and activity proceeds as normal.

Hence the partial recovery approach seems to increase the level of compositionality in our hybrid system. How exactly to implement this approach (for example, which partial recoveries to produce) is an area of ongoing research.

Heterogeneous Knowledge Representation

A strength of this hybrid executive system is that we can represent execution and repair knowledge in

a procedural way, a declarative way, or a combination, depending on the situation. This has proven to be useful in our domain. On the flip-side, this approach leads to a fair amount of duplicated knowledge between Exec and MIR. It would be interesting to develop an approach which permits maximal sharing of domain models between the two systems and yet still affords the representational power of our hybrid approach.

Dealing with Uncertainty

In this paper we have focused on the distinction between our procedural and deductive executives. Another important distinction relates to how each system represents uncertainty. The MI component of Livingstone views the world as a Partially Observable Markov Decision Process (POMDP), and maintains a distribution over likely world states. The MR component has access to the entire belief distribution, and so in principle can plan a sequence which maximizes expected utility given this information (Williams & Nayak 1996).

On the other hand, MI exports to Exec only the *most likely state* of the world. Exec acts as if this state were the true state, and responds quickly in the face of new information. Hence, Exec obeys the *rapid feedback principle* discussed by Schoppers (1995), and so most likely remains robust in the face of its unmodeled uncertainty. However, the lack of explicit communication of uncertainty and ambiguity between MI and Exec makes it difficult to write ambiguity resolution procedures in the Exec. At present, such procedures must be either deterministically inserted in the code (e.g., do a calibration experiment before thrusting the engine) or accessed via the interface with MR. Cassandra, Kaelbling, & Kurien (1996) provide a useful discussion of alternative strategies for managing uncertainty in planning and execution.

8 Conclusion

This paper has described the integration of procedural and deductive capabilities within a hybrid executive. While much work has been done on integrating planning and execution, comparatively little work has attempted to do so within a fast reactive loop or in the presence of concurrent activities. In addressing such an integration, we found we had to constrain or modify the component systems to address a number of technical problems. These problems included resource preemption, interacting concurrent recoveries, and non-compositionality of independent recoveries. The hybrid executive we

have developed addresses all these issues to some extent, and permits an extremely flexible and powerful representation of knowledge while still remaining robust and reactive.

Now that we have all this flexibility, a major challenge remains to understand how to take most advantage of it. Key issues include the following:

- Understanding the tradeoffs between representations of knowledge which are procedural, declarative, or hybrid.
- How to ensure consistency of knowledge across heterogeneous representations.
- Developing robust approaches to active sensing and active diagnosis within a hybrid executive.
- More integrated approaches to uncertainty management.

Lastly, it should be noted that our hybrid approach has evolved considerably over the last few years, based on lessons in the real spacecraft domain. We have now developed a hybrid between Livingstone and two different procedural execution systems: ESL (Gat 1996) and RAPS (Firby 1978). On the basis of this, we hope that our approach will be useful for integrating a wide variety of procedural and deductive executives. However, we found the explicit support for language extensions in ESL to be extremely useful for developing the new language constructs which enabled the strong integration discussed in this paper. This suggests language extension capabilities will make the job easier for other attempts to do a similar integration.

9 Acknowledgments

We acknowledge the contributions of other members of the DS1 Remote Agent team who have influenced the design of the hybrid execution architecture: Doug Bernard and Sandy Krasner.

References

Brown, G.; Bernard, D.; and Rasmussen, R. 1995. Attitude and articulation control for the cassini spacecraft: A fault tolerance overview. In *14th AIAA/IEEE Digital Avionics Systems Conference*.

Cassandra, A. R.; Kaelbling, L. P.; and Kurien, J. A. 1996. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. In Nourbakhsh, I., ed., *Proceedings of the AAAI Spring Symposium on Planning with Incomplete*

Information for Robot Problems. Palo Alto, CA: AAAI Press.

Fikes, R.; Hart, P.; and Nilsson, N. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):251–288.

Firby, R. J. 1978. *Adaptive execution in complex dynamic worlds*. Ph.D. Dissertation, Yale University.

Gat, E. 1996. ESL: A language for supporting robust plan execution in embedded autonomous agents. In Pryor, L., ed., *Procs. of the AAAI Fall Symposium on Plan Execution*. AAAI Press.

IJCAI. 1997. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Los Altos, CA: Morgan Kaufman Publishers.

Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Fox, M., and Zweben, M., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; and Williams, B. C. 1997a. An autonomous spacecraft agent prototype. In Johnson, W. L., ed., *Proceedings of the First Int'l Conference on Autonomous Agents*, 253–261. ACM Press.

Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997b. Robust periodic planning and execution for autonomous spacecraft. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (1997).

Schoppers, M. 1995. The use of dynamics in an intelligent controller for a space faring rescue robot. *Artificial Intelligence* 73(2):175–230.

Williams, B. C., and Nayak, P. P. 1996. A model-based approach to reactive self-configuring systems. In *Procs. of AAAI-96*, 971–978. Cambridge, Mass.: AAAI.

Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (1997).