A Hybrid Procedural/Deductive Executive For Autonomous Spacecraft [*]

Barney Pell [‡] Edward B. Gamble [§] Erann Gat [§] Ron Keesing [†] James Kurien [†]
William Millar [†] P. Pandurang Nayak [‡] Christian Plaunt [†] Brian C. 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 executive. A procedural executive provides sophisticated control constructs such as loops, parallel activity, locks, and synchronization which are used for robust schedule execution, hierarchical task decomposition, and routine configuration management. A deductive executive provides algorithms for sophisticated state inference and optimal failure recovery planning. The integrated executive enables designers to code knowledge via a combination of procedures and declarative models, yielding a rich modeling capability suitable to the challenges of real spacecraft control. The interface between the two executives ensures both that recovery sequences are smoothly merged into high-level schedule execution and that a high degree of reactivity is retained to effectively handle additional failures during recovery.

1 Introduction

We are developing the first on-board AI system to control an actual spacecraft (Bernard *et al.* 1998). 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 by near-Earth asteroid 3352 McAuliffe, Mars, and comet West-Kohoutek-Ikemura, taking pictures and sending back information to scientists on Earth. One key technology to be demonstrated is spacecraft autonomy, including robust plan execution (Pell *et al.* 1997b). Since aborting a plan and taking time to re-plan can cause the spacecraft to miss critical mission activities, execution of plans must be highly robust. Hence, the execution system must maintain spacecraft 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 (Muscettola 1994), robust multi-threaded execution (Gat 1996), and model-based diagnosis and reconfiguration (Williams & Nayak 1996; 1997).

Pell *et al.* (1997b) describes the approach we have taken to the automatic generation of robust plans, which incorporate flexibility to be used by the execution system in case problems or opportunities 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. A procedural executive provides sophisticated control constructs such as loops, parallel activity, locks, and synchronization which are used for robust schedule execution, hierarchical task decomposition, and routine configuration management. A deductive executive provides algorithms for sophisticated state inference and optimal failure recovery planning. The integrated executive enables designers to code knowledge via a combination of procedures and declarative models, yielding a rich modeling capability suitable to the challenges of real spacecraft control. The interface between the two executives ensures both that recovery sequences are smoothly merged into high-level schedule execution and that a high degree of reactivity is retained to effectively handle additional failures during recovery.

This paper discusses our domain, the component execution technologies, and the approach we took to integrating these technologies into a hybrid executive that supports the strengths of each while minimizing potentially negative interactions between the two systems. The paper is organized as follows. Section 2 discusses the spacecraft domain and requirements which influence our design. Section 3 describes

[*]This paper describes work partially performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract from the National Aeronautics and Space Administration.

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

[||]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.

our problem and hybrid approach to execution systems. Section 4 describes the capabilities in our procedural executive. Section 5 addresses the capabilities in the deductive executive. 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 spacecraft domain presents a number of challenges for robust plan execution.

2.1 High Reliability

A central requirement of spacecraft operation is *high reliability*, since spacecraft are expensive and often unique. Part of this high reliability is achieved through the use of reliable hardware. However, the harsh environment of space or the inability to test in all flight conditions can still cause unexpected hardware failures. When hardware failures or unexpected flight conditions do occur, the software system is required to compensate for such contingencies when possible. This requirement dictates the use of an executive with elaborate system-level fault protection capabilities. Such an executive can rapidly react to contingencies by retrying failed actions, reconfiguring spacecraft subsystems, or putting the spacecraft into a safe state to prevent further, potentially irretrievable, damage.

2.2 Concurrent Temporal Processes

Many devices and systems must be controlled, leading to multiple threads of complex activity. These concurrent processes must be coordinated to control for 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 short time period of observability.

2.3 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 physical 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 system. In particular, resetting the engine gimbal, to fix an engine problem, causes temporary loss of control of the solar panels. Thus, fixing one problem can cause new problems to arise. To avoid this, the recovery system needs to take into account global constraints from nominal schedule execution, rather than just making local fixes in an incremental fashion. 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.

3.1 The Problem

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

On the one hand, execution requires reactivity, time-sensitivity, and sophisticated control constructs such as 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 (Firby 1978).

However, this strict procedural approach has its limitations — it is hard to procedurally encode optimal choices in all, possibly degraded, situations. Specifically, execution requires choosing component configurations with different capabilities and costs. Similarly, robust recovery may require novel combinations of actions in order to trade off costs and benefits. For example, the propulsion system on the Cassini spacecraft (Brown, Bernard, & Rasmussen 1995) has a complex set of valves, including explosive *pyro* valves which can change states only once, and ordinary valves with varying amounts of wear and tear. It is difficult to procedurally express the right valve choices to redirect fluid flow while minimizing costs and risks in all possible situations.

On the other hand, a deductive executive of the form developed by Williams & Nayak (1996) can reason efficiently about such tradeoffs using declarative models of the costs and benefits of configurations and recoveries. Furthermore, the compositional nature of such models allows compact representations of the costs and benefits of each possible choice. Finally, deductive executives have sophisticated state inference algorithms, supporting the identification of hidden state, failed sensors, and multiple faults. However, declarative models can 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 temporal logic which does not explicitly model metric time or execution loops. Thus it is hard to encode knowledge like:

> 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]

3.2 Hybrid Approach

From this we see that the procedural and deductive approaches to execution have complementary strengths and weakness. Hence, our approach is to develop a hybrid executive, as follows:

- Use a procedural executive for timing, 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/benefit analysis.

[1]The 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 be switched off while the surge is happening, and then switched on again.

- Work out clear interfaces between 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. For example, while routine configuration management can be handled either procedurally or declaratively, we have chosen to handle it procedurally. In our treatment, the procedural executive draws on the planning capability of the deductive executive by using it as a *recovery expert*, sending it a set of global constraints that ensure that the resulting recovery plans can be integrated within the current execution context.

In the next sections we describe the procedural executive and the 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 called Executive Support 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. Some of these constructs are summarized later in this section. An executive also needs a source of state update knowledge. In NMRA, Exec benefits from being insulated from the hardware details by relying on the results of the mode identification (MI) component of MIR (see Section 5).

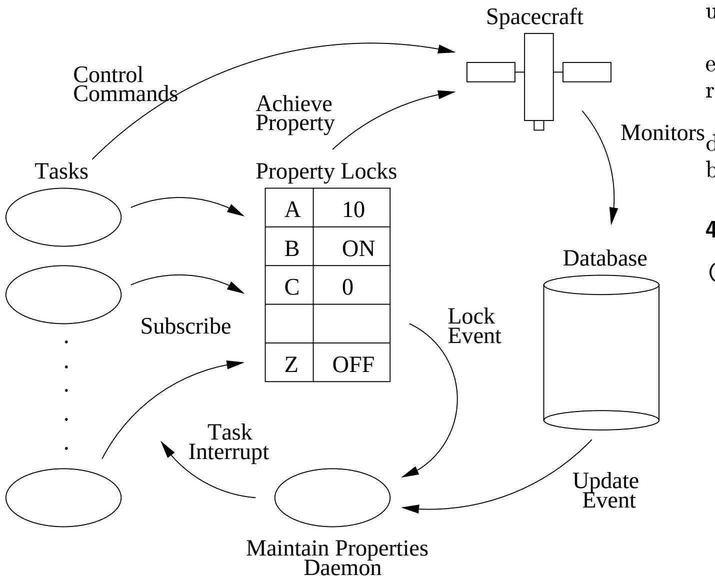

Figure 1: Procedural Executive Resource Manager

The executive manages a set of concurrent control tasks, as shown in Figure 1. Each control task requires a set of *resources*, or *properties*, to be established and maintained over some period of time. For example, the activity of taking pictures with a camera requires that the camera is on and functional. If some other activity requires the camera to be off, these two activities *compete* for the resource of controlling the camera's power state. The executive must achieve, maintain, and monitor properties required for each task, and resolve task resource conflicts.

A task is represented at run-time by an independent execution *thread*. Threads communicate with other theads directly via *signals*, or indirectly via changes to a *database*. Receipt of a signal or notification of a change to the database are examples of *events*.

Each activity uses the `(with-maintained-properties)` construct to declare those properties that it requires maintained over its interval of execution. In this way, Exec understands the constraints which support the entire current execution context. When a property is achieved and reserved for a task, it is said to be *locked* until the task relinquishes it, so that other tasks will not be permitted to violate that property. Of course, the locks reflect properties true in the current state, and sometimes these properties can change despite the best efforts of the software system to maintain them. For example, switches on a spacecraft sometimes change state accidentally. In this case, we describe the properties as *lost* or *violated*, and the tasks requiring them as *unsupported*.[2]

In the event that some property is lost or otherwise unachievable without the help of a recovery expert, Exec suspends the unsupported threads, formulates a query based on the active constraints, and uses the `automatic-recoveries` thread to send the query off to the recovery expert (in this case, MIR).

When the recovery expert returns an action, Exec performs the action and then re-activates any suspended threads which may now be supported. The threads then attempt to re-establish their maintained conditions. 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 exceeds a threshold.

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

We now elaborate on some of the key constructs we have developed within the procedural executive that support the behavior described above.

4.1 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 the 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).

4.2 Maintained Properties

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

[2]Note that property locks can serve a role similar to typical locks in multi-threaded systems, such as semaphores and mutexes. However, there is a major difference since these property locks are database-relative, and can hence be "taken" by the outside world changing. Note also that naive use of property locks can result in deadlock, just as occurs with standard locks in multi-threaded operating systems.

- If *properties* are all currently true, *body* is executed.
- If *properties* are false, the executive tries to achieve them first.
- 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 the enclosing context of *body* choose the response.

4.3 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
   :powered-thru :power_bus_1
   :switched-thru :fsc_camera_sw1
   :ready-state ((:health_state :ok)
                 (:power_state :on)))
```

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. Based on the camera definition above,

```
 (with-selected-device :camera (take-pictures))
```

would select a camera (say *camera_A*), achieve its ready state of being powered on and healthy, and then take pictures within a context that ensures that the health and power of the camera are maintained throughout picture taking.

4.4 Recovering failed properties

In the case where a maintained property is lost (for example, device switch flips off unexpectedly or the engine performs an automatic shutdown), the enclosing context of the `(with-maintained-properties)` form determines the appropriate response. If no response is defined for the enclosing context, then the form fails.

`(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. Note that this is only one way to create an enclosing context to handle the lost properties notification. For example, a thread could establish its own local recovery expert, or decide to try alternative methods if properties are lost, rather than waiting for a automatically generated recovery.

4.4.1 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 effect of signaling `recovery-event` is to wake up all threads who were suspended waiting for a property which was restored (possibly as a result of the recovery action). Each awakened thread then retries the body, attempting to re-establish all their 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.

If 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

The deductive executive can be viewed as a discrete model-based controller that attempts to keep the spacecraft state on a trajectory that achieves a set of high-level input properties (analogous to the set-point of a continuous controller). In the NMRA architecture, the dedective executive is also referred to as MIR reflecting that control is achieved through mode identification (the sensing component) and mode reconfiguration (the actuation component).

MIR is model-based in the sense that it uses a single declarative, compositional model of the spacecraft to support all of its capabilities. MIR views each component as a finite state machine, and the entire spacecraft as concurrent, synchronous state machines. Nodes in the graph represent behavioral *modes*, and arcs represent possible *transitions* among modes, some exogenous, some commandable. Modes partition the state space of the component, and are specified using well-formed formulae in a propositional language.

Mode identification (MI) involves tracking the most likely trajectory of the spacecraft state by observing all commands that are sent to the spacecraft and monitoring information from spacecraft sensors. Each point in a trajectory consists of the current behavioral mode of each component in the spacecraft. Components include both hardware devices and lower-level software modules. With modes identified, more detailed component state information is available at the propositional level.

MI provides a service for tracking and reporting state changes to external software modules as they occur. The idea is that external modules will typically be interested only in higher-level properties (and corresponding higher-level events) involving spaceraft state, rather than the finer grained view available to MI. These abstract properties are naturally defined as well-formed formulae, and are easily tracked using MI's inference capabilities. In the NMRA architecture, MI's state update service is an integral part of the interface between MIR and Exec.

Mode reconfiguration (MR) involves generating a sequence of actions that moves the spacecraft from its most likely current state to a new state that achieves a desired set of properties. MR is comprised of two stages. First, the requested set of properties to be achieved is used to generate a specific goal state for each of the spacecraft's components. Second, a sequence of actions that move the spacecraft from the current state to the goal state is incrementally generated. We refer to this second stage as model-based reactive planning (MRP). The sequence may be empty meaning that no action is necessary, or sequence generation may fail meaning that no reconfiguration plan could be found. Each action in the sequence is a primitive operator from the perspective of MIR's models. When MIR functions as a stand-alone deductive executive, each primitive operator corresponds to a command directly executable by an external software module. In the NMRA architecture, Exec specifies the desired properties of the goal state and primitive operators in the action sequence are bound to Exec procedures.

MIR uses algorithms adapted and extended from model-based diagnosis (de Kleer & Williams 1987; 1989) to provide the above functionality. The main idea behind model-based diagnosis is to identify the set of possible component states in a system given models and observations of each component in the system. In many systems, especially spacecraft, there may be inadequate information in the models and observations to uniquely identify every component's state at all times. The approach is thus to select the *most likely* component configuration from amongst those that are *consistent* with the models and observations.

The primary workhorse in the deductive executive is an extremely efficient conflict-directed best-first search algorithm (Williams & Nayak 1996). The algorithm is exploited by MI to identify the most likely component configuration consistent with models and observations, and by MR to select a specific goal state having a specified set of properties. Additionally, a recent approach to MRP (Williams & Nayak 1997) exploits the algorithm at compile time to compile away irrelevant information in system models in support of efficient planning. Such reuse of algorithms and system models across MIR's capabilities is a signature of the model-based approach, and greatly simplifies the development and maintenance of our deductive executive.

6 Integration

Having described the procedural and deductive executives, we now discuss how we combined these systems in the NMRA architecture to form an integrated hybrid executive. Recall that we exploit the procedural executive (Exec) for schedule execution, hierarchical task decomposition, and routine configuration management, while the deductive executive (MIR) is used both for state inference and failure response.

Here we make explicit that the communication interface between Exec and MIR consists of the following: state updates from MIR to Exec, recovery requests from Exec to MIR, and recovery actions from MIR to Exec. Both state updates and recovery requests are represented as well-formed formulae in a propositional language shared between Exec and MIR. Recovery actions are instantiations of Exec's generic procedures.

To support state updates, MIR continually tracks the most likely state of the spacecraft and informs Exec of changes to any higher-level property it wants tracked. Exec uses this state information to make task decomposition and configuration management decisions, and to determine the truth of properties needed by various threads of execution. Exec procedures are generally written to exploit MIR by allowing it to perform most inferences about spacecraft state that may be required. The properties to be tracked for Exec by MIR are agreed upon at compile time, but we note that the interface can be extended naturally to allow the notion of registering tracked properties on the fly; such run-time flexibility would allow for more efficient communication during critical mission phases, and enable Exec activities to dynamically declare their own interface with MIR to improve modularity.

Exec also views MIR as a recovery expert. As events occur in Exec's schedule, it provides MIR with the current set of properties that must be maintained to support all active threads. At the time of invocation, some of these properties will be true and some may be false. Using its declarative models and knowledge of the current state, MIR generates an action sequence that is expected to move the spacecraft to a goal state in which all the requested properties are achieved. MIR provides the first action in this sequence to Exec. Exec then executes this action and waits for state updates from MIR to determine the status of its required properties. The recovery interaction repeats with MIR until either all desired properties are achieved or MIR indicates that it can find no sequence to achieve those properties.

Three points are worth noting about the recovery interface. First, note that MIR sends only the first action in the recovery sequence. This improves the reactivity of the hybrid executive in two ways: Exec is free to make finer grained recovery requests to reflect any changes in the status of schedule execution since the previous request, while MIR is free to factor any asynchronous spacecraft state changes that may have occurred into its next recovery plan. Achieving this level of reactivity would be somewhat more difficult if the Exec were expected to robustly execute a full plan returned from MIR, for either the plan would then have to be much larger to reflect all contingencies or Exec would have to encode the robustness into the primitive procedures over which MIR reasons.

Second, treating recovery actions as instances of generic procedures fully exploits the representational strengths of both systems. In practice, a natural modeling approach that addressed both representional convenience and efficiency was to encapsulate all issues related to metric time and iteration inside Exec's procedural constructs. This was natural, for instance, in the case of the downlink example provided in Section 3.

Third, note that when used as a stand-alone configuration system, MIR is free to generate any sequence of actions resulting in a state with the requested properties. However, as part of the hybrid executive, properties requested during recovery are viewed as constraints on the entire recovery plan, not just the goal state; this means that MIR must not generate a recovery plan that is expected to deviate from a requested property. Depending on the approach to MRP that one adopts, this places additional computational requirements on the reactive planner that may require one to give up optimality or efficiency guarantees; this is indeed the case for the approach used in (Williams & Nayak 1997), for example. Combined with the requirement on Exec to include all required properties as part of a recovery request, this restriction on MIR ensures that recovery sequences are smoothly merged into nominal schedule execution, resolving the problems of resource preemption and interacting recoveries discussed in Section 2.

7 Discussion and Future Work

In this section we discuss ongoing issues and limitations in our current hybrid executive and indicate future work.

7.1 Compositionality and Modularity

A major design goal within the NMRA is to develop modular, compositional representations of spacecraft subsystems. A subtle limitation violating this goal exists in our current recovery framework; it arises in the context of multiple failures, even when they occur in otherwise independent subsystems.

Consider two independent subsystems, managed seperately by two Exec activities. Suppose one subsystem can be recovered if it fails, and the other cannot. In the event of independent failures in each subsystem, the recovery framework would procede through two seperate recovery attempts and result predictably in the recovery of one subsystem. However, should those same failures instead occur in sufficiently close temporal proximity, MIR would report the failures to the Exec *simultaneously*. The Exec would then form a recovery request to MIR asking for the recovery of the *conjunction* of the two failed properties of interest. MIR would then be forced to report that no such recovery is possible (since only one of the properties is recoverable) and the Exec activities managing the independent subsystems would both fail, resulting in the recovery of neither subsystem.

The standard response to this problem is to emphasize that this limitation only arises in the case of simultaneous, independent failures. For most missions, such events are deemed sufficiently unlikely that they are considered acceptable risks and beyond the scope of current fault protection systems. It is worth noting that this risk assessment is based in part on another limitation of current fault protection frameworks: the mindset within the spacecraft community is that unlikely hardware failures are less likely than a design flaw in a complex fault protection system that attempts to support these unlikely failures. Our methods aim to address this general concern first and foremost by simplifying the design of robust execution systems to enable broader fault coverage. We view modularity and compositionality as key requirements of a simple design.

The solution is to augment the recovery framework to enable consideration of partial recoveries, rather than attempting an all-or-nothing recovery. The open design issue is to understand whether this is best accomplished with modifications to Exec or MIR. In the former case, Exec could be augmented to formulate a series of independent partial recovery requests that would collectively achieve total feasible recovery, giving priority to the most urgent activities. The intuition here is to have Exec be more clever in asking for only what it needs, though this would currently require access to system models stored in MIR. Alternatively, MIR could generate recovery plans that satisfy a maximal subset of the requested properties, though in practice this would require additional communication between Exec and MIR to allow Exec to specify its preferences. These approaches are complementary, and striking a proper balance between them is an area of ongoing research.

7.2 Heterogeneous Knowledge Representation

A strength of our hybrid executive system is that we can represent execution and repair knowledge in a procedural way, a declarative way, or a combination thereof, depending on the situation. This has proven to be useful in our domain. On the flip-side, this approach can lead to a fair amount of duplicated knowledge between Exec and MIR. We are currently developing an approach to permit maximal sharing of domain models across the two systems that still affords the representational power and convenience of our hybrid approach. Note that this sharing of system models also supports the partial recovery issue addressed above by enabling Exec to access system models during formulation of partial recovery sequences.

7.3 Dealing with Uncertainty

Ambiguity management is a critical issue in spacecraft operations, primarily due to limitations in the number and type of onboard sensors and the possibility of sensor failures. Recall that MIR currently tracks only the most likely trajectory of the spacecraft, a restriction driven primarily by the severely limited onboard computation available to it (10% of a 20MHz CPU on DS-1). MIR deals with ambiguity by assuming a worst-case scenario. For example, if there is ambiguity as to whether a device has failed or a communication path to that device has failed, MIR assumes that both have failed. Although this construction of a worst-case trajectory works well in the case of the DS-1 models, one can construct models for which the worst-case scenario leads to suboptimal recoveries and unsound conclusions. We are working an approach that allows MIR to track a small set of the most likely trajectories to deal more cleanly with ambiguity in an efficient manner.

Recall further that 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 is more likely to remain 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 hard-wired in the code (e.g., do a calibration experiment before thrusting the engine) or accessed exclusively via the interface with MR. We are pursuing an approach to active testing wherein Exec and MIR cooperate to synthesize optimal sequences from system models that resolve ambiguity in a manner that preserves spacecraft safety and non-renewable resources.

8 Related Work

This paper has described the integration of procedural and deductive capabilities within a hybrid executive. This section discusses related work and addresses procedural reasoning systems that provide support for deduction, deductive reasoning systems that provide support for reaction, hybrid action description languages, and systems that cleanly separate a deductive planning or inference component from a procedural execution component.

Like our Exec, RAPS (Firby 1978) is centered around procedural reasoning, but provides language features to express deductive state inference (in the form of `memory-rules`) and to incorporate the results of deductive problem-solving systems (in the form of `problem-solvers`). RAPS also provides constructs to indicate resource locks for thread synchronization, but these constructs are used only at the lowest level of the system.

PRS (Georgeff & Lansky 1987) is also similar to our Exec in that it provides a language based around procedural reasoning and it has been applied to support diagnosis (Georgeff & Lansky 1986) and plan execution (Georgeff, Lansky, & Schoppers 1987). PRS also provides support for procedures to perform meta-level reasoning about execution context (Ingrand & Georgeff 1990) and some constructs to express resource usage to prevent harmful task interactions (e.g., the `require` construct).

Our hybrid executive extends the capabilities of these systems (and similar procedural reasoners such as RPL (McDermott 1993) and APEX (Freed & Remington 1997)) in two major ways. The first is to provide explicit access to deductive model-based reasoning for diagnosis and planning. The second is to extend resource locks into a task-level construct and to provide a way to use them to constrain the results of deductive inference.

While Exec, RAPS and PRS may be viewed as procedural reasoning systems with deductive attachments, a large body of work in automated reasoning has focused on deductive reasoning systems with procedural attachments (Genesereth & Nilsson 1987). Most of this work focuses on using procedures to support inference, rather than on describing action in a dynamic environment. However, researchers have recently begun exploiting the ability to view logical systems like Prolog (Clocksin & Mellish 1981) through both an *operational* and a *denotational* semantics to create logical descriptions of procedures which can support both procedural and deductive reasoning in the presence of a changing environment. Example systems include Golog (Levesque *et al.* 1997; de Giacomo, Lesperance, & Levesque 1997) and InterRAP (Muller & Pischel 1994).

Estlin, Chien, & Wang (1997) describe a hybrid approach to action descriptions for planning systems that integrates Hierarchical Task Network (HTN) planning, which can be viewed as a procedural representation, with operator-based planning, which deduces action sequences from first principles. Also related is the OSCAR architecture (Pollock 1998), which integrates planning and reasoning activities within a general-purpose defeasible reasoner.

Perhaps the most typical approach to developing a hybrid system is to develop separate components for both styles of reasoning and define a clear interface to support the interaction. Much of the research on integrating planning and execution (Wilkins *et al.* 1995; Bonasso *et al.* 1997; Hayes-Roth 1995; Simmons 1990; Currie & Tate 1991; Pell *et al.* 1998, for example) takes this approach. Whereas these systems generally treat the planner and executive as functioning on widely different time-frames, our approach exploits fast deduction to provide these capabilities within the reactive execution loop itself.

In terms of separate components for procedural execution and deductive state inference (as opposed to planning), Ogasawara (1991) describes a hybrid architecture based on Bayesian networks and decision-theory for state inference, where the results of inference can be used by a system executing high-level procedures. The Touring Machine architecture (Ferguson 1992) also provides a separate capability for deductive world modeling that informs the activities of a procedural executive.

9 Conclusion

This paper has described the integration of procedural and deductive capabilities within a hybrid executive. While there has been much research on integrating planning or state inference with execution and on incorporating procedures within deductive systems or vice-versa, 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 this flexibility, a major challenge remains to understand how to take most advantage of it. Key issues include the following:

- Understanding the tradeoffs between knowledge representations that 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 hybrids between Livingstone (Williams & Nayak 1996) 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.

10 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. We thank John Bresina, Greg Dorais, Michael Lowry, Rich Washington, and the anonymous reviewers for useful comments on drafts of this text.

References

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

[2] Bonasso, R. P.; Kortenkamp, D.; Miller, D.; and Slack, M. 1997. Experiences with an architecture for intelligent, reactive agents. *JETAI* 9(1).

[3] 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.*

[4] Clocksin, W. F., and Mellish, C. S. 1981. *Programming in Prolog.* Springer-Verlag: Berlin, Germany.

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

[6] de Giacomo, G.; Lesperance, Y.; and Levesque, H. 1997. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Procs. of IJCAI-97*, 1221–1226.

[7] de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130. Reprinted in (Hamscher, Console, & de Kleer 1992).

[8] de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proceedings of IJCAI-89*, 1324–1330. Reprinted in (Hamscher, Console, & de Kleer 1992).

[9] Estlin, T. A.; Chien, S. A.; and Wang, X. 1997. An argument for a hybrid HTN/operator-based approach to planning. In *Procs. of the Fourth European Conference on Planning.*

[10] Ferguson, I. A. 1992. *Touring Machines: An Architecture for Dynamic, Rational, Mobile Agents.* Ph.D. Dissertation, Computer Laboratory, University of Cambridge.

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

[12] Freed, M., and Remington, R. 1997. Managing decision resources in plan execution. In *Procs. of IJCAI-97*, 322–326.

[13] 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.

[14] Genesereth, M. R., and Nilsson, N. J. 1987. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann: Los Altos, CA.

[15] Georgeff, M. P., and Lansky, A. L. 1986. A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle. Technical Note 375, Artificial Intelligence Center, SRI International.

[16] Georgeff, M. P., and Lansky, A. L. 1987. Procedural knowledge. Technical Report 411, Artificial Intelligence Center, SRI International.

[17] Georgeff, M. P.; Lansky, A. L.; and Schoppers, M. J. 1987. Reasoning and planning in dynamic domains: An experiment with a mobile robot. Technical Report 380, Artificial Intelligence Center, SRI International.

[18] Hamscher, W.; Console, L.; and de Kleer, J. 1992. *Readings in Model-Based Diagnosis.* San Mateo, CA: Morgan Kaufmann.

[19] Hayes-Roth, B. 1995. An architecture for adaptive intelligent systems. *Art. Int.* 72.

[20] Ingrand, F. F., and Georgeff, M. P. 1990. Managing deliberation and reasoning in real-time ai systems. In *Procs. DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, 284–291.

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

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

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

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

[25] Ogasawara, G. H. 1991. A distributed, decision-theoretic control system for a mobile robot. *ACM SIGART Bulletin* 2(4):140–145.

[26] 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.

[27] Pell, B.; Gat, E.; Keesing, R.; Muscettola, N.; and Smith, B. 1997b. Robust periodic planning and execution for autonomous spacecraft. In *Procs. of IJCAI-97.*

[28] Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; and Williams, B. C. 1998. An autonomous spacecraft agent prototype. *Autonomous Robotics* 5(1). To Appear.

[29] Pollock, J. L. 1998. Planning agents. In Rao, A., and Wooldridge, M., eds., *Foundations of Rational Agency.* Kluwer.

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

[31] Simmons, R. 1990. An architecture for coordinating planning, sensing, and action. In *Procs. DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, 292–297.

[32] Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1995. Planning and reacting in uncertain and dynamic environments. *JETAI* 7(1):197–227.

[33] 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.

[34] Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Procs. of IJCAI-97.*