

Abstract Resource Management in an Unconstrained Plan Execution System

Erann Gat
JPL MS 525-3660
4800 Oak Grove Drive
Pasadena, CA 91109
(818) 354-4674
gat@jpl.nasa.gov

Barney Pell
NASA Ames Research Center
Mountain View, CA
(415) 604-3361
pell@ptolemy.arc.nasa.gov

Abstract—We describe the abstract resource management mechanism in ESL (Execution Support Language [3]). ESL is the implementation substrate for the New Millennium Remote Agent Smart Executive [9], part of a NASA program to demonstrate autonomous control of an unmanned spacecraft scheduled to launch in 1998. The executive is responsible for robust plan execution in the face of unexpected run-time contingencies. Part of this task requires run-time management of the spacecraft's configuration, whose component states are modeled as abstract resources. In this paper we describe the ESL constructs for managing these abstract resources. The resource management facilities in ESL are similar to the constraint management constructs in RAPs [2]. The major contribution in this paper is the implementation of these facilities in an unconstrained execution substrate implemented as an extension to a standard programming language (in this case, Common Lisp) rather than within a constrained self-contained plan execution language. This turns out to significantly simplify complex programming tasks.

The main technical problem in a resource management system is designing a representation that allows automatic determination of when conflicts exist. In its full generality this becomes a full-blown planning problem, and therefore an impractical strategy for a reactive executive. Instead, we model abstract resources as *properties*, logical assertions whose final values are guaranteed unique. When two properties are identical but for their final value then they are in conflict. This paper describes the ESL constructs and mechanisms for scheduling tasks so that they do not attempt to achieve conflicting properties simultaneously, and for invoking external recovery mechanisms for restoring properties to their desired states when forced away from those states by unexpected contingencies.

TABLE OF CONTENTS

1. INTRODUCTION
2. ESL REVIEW
3. THE PROBLEM
4. PROPERTY LOCKS
5. APPLICATIONS
6. SUMMARY

1. INTRODUCTION

New Millennium is a NASA program for demonstrating advanced technologies for unmanned space exploration. The first New Millennium Mission is called Deep-Space 1 (DS1), and among other advanced technologies to fly on this mission is an autonomy architecture called the Remote Agent (RA) [8]. The Remote Agent consists of three major components, a planner-scheduler, a model-based diagnosis system, and an executive, which is responsible for plan execution, managing run-time contingencies, and overall coordination of spacecraft activities. The executive is written in a specially designed language, ESL [3,6], which provides advanced control constructs to simplify the task of programming autonomous reactive control systems.

This paper describes the design and implementation of one of ESL's feature sets: abstract resource management (ARM). An abstract resource is a component of the state of a system. Abstract resources include consumable resources like fuel, renewable resources like electrical power and data storage, and discrete states of spacecraft components such as power states, and operational modes. The current ARM features of ESL address only discrete-state resources.

The general problem of run-time resource management is equivalent to the planning problem and is thus intractable. We approach the problem here not as a theoretical problem but as a software engineering problem: what sorts of language constructs are required to enable human programmers to effectively encode run-time knowledge

about resource management? Our approach is based on the idea of *memory properties* [2], logical assertions whose last element is guaranteed to be unique. Memory properties provide a useful structure both for determining when resource conflicts exist, and for recovering from conflicts.

We begin with a brief review of ESL's basic structure and features.

2. ESL REVIEW

ESL (Execution Support Language) is a language for encoding execution knowledge in embedded autonomous agents. It is designed to be the implementation substrate for the sequencing component of a three-layer architecture such as 3T [1] or ATLANTIS [4]. The sequencer in such an architecture coordinates the actions of a reactive controller, which controls the agent's actions, and a deliberative component, which generates plans and performs other high-level computations. The sequencer must be able to respond quickly to events while bringing potentially large quantities of information — both knowledge and run-time data — to bear on its decisions. An implementation substrate for such a system should also be able to deal with a variety of different strategies for assigning responsibilities to the various layers, from mostly reactive strategies, to ones where the planner is the prime mover.

ESL is similar in spirit to RAPs [2], RPL [8], and RS [7], and its design owes much to these systems. Unlike its predecessors, ESL aims for a more utilitarian point in the design space. ESL was designed primarily to be a powerful, flexible, and easy-to-use tool, not to serve as a representation for automated reasoning or formal analysis (although nothing precludes its use for these purposes). ESL consists of several sets of loosely coupled features that can be composed in arbitrary ways. It is currently implemented as a set of extensions to Common Lisp.

The following sections provide a brief overview of some of the major feature sets in ESL. Most of these are used by the resource management mechanism described in section 4. For a complete description of the language see the ESL User's Guide [6].

Contingency Handling

The contingency-handling constructs of ESL are based on the concept of *cognizant failure*, which is a design philosophy that states that systems should be designed to detect failures when they occur so that the system can respond appropriately. This approach presumes that the multiple possible outcomes of actions are easily categorized as success or failure. (It also assumes that failures are inevitable.) This approach can be contrasted with approaches such as universal plans [10] where multiple outcomes are all treated homogeneously. Our experience has been that the cognizant-failure approach provides a good reflection of human intuitions about agent actions.

Basic constructs —The two central contingency-handling constructs of ESL are a means of signaling that a failure has

occurred, and a means of specifying a recovery procedure for a particular type of failure. These constructs are:

```
(FAIL cause . arguments)
```

```
(WITH-RECOVERY-PROCEDURES  
 (&rest recovery-clauses)  
 &body body)
```

The FAIL construct signals that a failure has occurred, and WITH-RECOVERY-PROCEDURES sets up recovery procedures for failures. A call to FAIL is equivalent to a call to an active recovery procedure (i.e. one whose restarts limit has not been reached). Recovery procedures have dynamic scope.

The syntax for a recovery clause is:

```
(cause &key retries . body)
```

or

```
((cause . args) &key retries . body)
```

In the first case any arguments in a FAIL statement which transfers control to the recovery procedure are discarded. In the second case arguments are lexically bound to ARGS. Excess arguments are discarded, and missing arguments default to nil. The optional keyword argument RETRIES specifies the maximum number of times that particular recovery procedure can be invoked during the current dynamic scope of the WITH-RECOVERY-PROCEDURES form. RETRIES defaults to 1. The value of RETRIES can be the keyword :INFINITE, with the obvious results.

Within the BODY of a recovery procedure the special form (RETRY) does a non-local transfer of control (a throw) to the BODY of the WITH-RECOVERY-PROCEDURES form of which the recovery procedure is a part, and the special form (ABORT &optional result) causes RESULT to be immediately returned from the WITH-RECOVERY-PROCEDURES form.

A recovery procedure for cause :GENERAL-FAILURE is applicable to a failure of any cause. It is possible to generalize this mechanism to a full user-defined hierarchy of failure classes, but so far we have not found this to be necessary.

The scope of a set of recovery procedures is mutually recursive in the manner of the Lisp LABELS construct, or Scheme LETREC. That is, the scope of a recovery procedure includes the recovery procedure itself, and all other recovery procedures that are part of the same WITH-RECOVERY-PROCEDURES form. Failures are only propagated upwards when no recovery procedures for a given failure exist within the current WITH-RECOVERY-PROCEDURES form, or when all the retries for that failure have been exhausted. For example, the following code will print FOO BAZ FOO BAZ, and then fail with cause :FOO.

```
(with-recovery-procedures
```

```
( (:foo :retries 2
  (print 'foo) (fail :baz))
  (:baz :retries 2
  (print 'baz) (fail :foo)) )
(fail :foo))
```

Cleanup procedures —It is often desirable to insure that certain actions get taken "if all else fails" and the execution thread exits a certain dynamic context with a failure. For example, one might want to insure that all actuators are shut down if a certain procedure fails and the available recovery procedures can't deal with the situation. Such a procedure is called a cleanup procedure, and is provided in ESL using the following construct:

```
(WITH-CLEANUP-PROCEDURE cleanup
 &body body)
```

This construct executes BODY, but if BODY fails, CLEANUP is executed before the failure is propagated out of the WITH-CLEANUP-PROCEDURE form. This construct is similar to the Lisp UNWIND-PROTECT construct except that the cleanup procedure is only executed if BODY fails. (Because ESL is implemented on top of Common Lisp, UNWIND-PROTECT is also available for implementing unconditional cleanup procedures.)

Goal Achievement

Decoupling of achievement conditions and the methods of achieving those conditions is provided by the ACHIEVE and TO-ACHIEVE constructs. The syntax for these constructs is:

```
(TO-ACHIEVE condition . methods)
(ACHIEVE condition)
```

Each METHOD is a COND clause. For example:

```
(to-achieve (device-ready)
 ( (eq (device-power-state) :off)
  (turn-device-on) )
 ( (eq (device-health-state) :single-
  event-upset)
  (reset-device) )
 ( (eq (device-health-state) :permanent-
  failure)
  (fail :device-permanently-failed) ))
```

The TO-ACHIEVE construct is somewhat analogous to the RAP METHOD clause in that it associates alternative methods with conditions under which those methods are appropriate.

Task Management

Events —ESL supports multiple concurrent tasks. Task synchronization is provided by a data object type called an *event*. A task can wait for an event, at which point that task

will block until another task signals that event. The constructs are straightforward:

```
(WAIT-FOR-EVENTS events
 &optional test)
(SIGNAL event &rest args)
```

A task can wait on multiple events simultaneously; it becomes unblocked when any of those events are signaled. Also, multiple tasks can simultaneously wait on one event. When that event is signaled, all the waiting tasks are unblocked simultaneously. (Which task actually starts running first depends on the task scheduler.)

If arguments are passed to SIGNAL-EVENT those arguments are returned as multiple values from the corresponding WAIT-FOR-EVENT. If WAIT-FOR-EVENTS is provided an optional TEST argument, then the task is not unblocked unless the arguments passed to SIGNAL answer true to TEST (i.e. TEST returns true when called on those arguments).

Checkpoints —ESL tasks are themselves first-class data objects which inherit from event. Thus, tasks can be waited-for and signaled. However, because tasks have a linear execution thread it is desirable to slightly modify the semantics of an event associated with a task. Normal events do not record signals; a task waiting on an event blocks until the next time the event is signaled. However, a task waiting for another task should not block if the other task has already passed the relevant point in the execution thread. (For example, if task T1 starts waiting for task T2 to end after T2 has already ended it should not block.) Thus, ESL provides an additional mechanism called a checkpoint for signaling task-related events. Signaling a checkpoint is the same as signaling an event, except that a record is kept of the event having happened. When a checkpoint is waited-for, the record of past signals is checked first. In order to disambiguate checkpoints, an identifying argument is required. Thus we have the following constructs:

```
(CHECKPOINT-WAIT task id)
(CHECKPOINT id)
```

CHECKPOINT-WAIT waits until checkpoint ID has been signaled by task TASK. CHECKPOINT signals checkpoint ID in the current task. There is a privileged identifier for signaling a checkpoint associated with the end of a task. This checkpoint is automatically signaled by a task when it finishes. To wait for this privileged identifier there is an additional construct, WAIT-FOR-TASK, which is simply a CHECKPOINT-WAIT for the task-end identifier.

Task nets —ESL provides a construct called TASK-NET for setting up a set of tasks in a mutually recursive lexical context. The syntax is:

```
(TASK-NET [:allow-failures]
 (identifier &rest body)
 (identifier &rest body)
 ...)
```

The bodies in a TASK-NET are run in parallel in a lexical scope in which the identifiers are bound to their corresponding tasks. The TASK-NET form itself blocks until all its children finish. Unless the optional :ALLOW-FAILURES keyword is specified, if one subtask in a task net fails the other tasks are immediately aborted and the whole TASK-NET construct fails. There is also an OR-PARALLEL construct which finishes when any one of its subtasks finishes successfully, or all of them fail.

For example, the following code prints 1 2 3 4:

```
(TASK-NET
  (t1 (print 1)
      (checkpoint :cp)
      (checkpoint-wait t2 :cp)
      (print 3))
  (t2 (checkpoint-wait t1 :cp)
      (print 2)
      (wait-for-task t1)
      (print 4)))
```

Guardians —One common idiom in agent programming is having a monitor task which checks a constraint that must be maintained for the operation of another task. We refer to the monitoring task as a *guardian* task. The relationship between a guardian and its associated main task is asymmetric. A constraint violation detected by the guardian should cause a cognizant failure in the main task, whereas termination of the main task (for any reason) should cause the guardian to be aborted. This asymmetric pair of tasks is created by the following form:

```
(WITH-GUARDIAN guardform failform
 &body body)
```

WITH-GUARDIAN executes BODY and GUARDFORM in parallel. If body ends, GUARDFORM is aborted. If GUARDFORM ends, then the task executing body is interrupted and forced to execute FAILFORM (which is usually a call to FAIL).

For example, the following code operates a widget while monitoring the widget in parallel. If MONITOR-WIDGET returns, then OPERATE-WIDGET will fail cognizantly.

```
(with-guardian (monitor-widget)
               (fail :widget-failed)
  (operate-widget))
```

Logical Database

A logical database is provided as a modular functionality in ESL. The major constructs supporting this database are ASSERT and RETRACT, for manipulating the contents of the database, DB-QUERY for making queries, and WITH-QUERY-BINDINGS, which establishes a dynamic context for logical variable bindings and continuations. The syntax for WITH-QUERY-BINDINGS is:

```
(WITH-QUERY-BINDINGS query [:inherit-
                             bindings] . body)
```

Within a WITH-QUERY-BINDINGS form a call to NEXT-BINDINGS calls the binding continuation, i.e. it causes a jump to the start of BODY with the next available bindings for QUERY. If there are no more bindings, NEXT-BINDINGS fails with cause :NO-MORE-BINDINGS. (If there were no bindings to begin with the WITH-QUERY-BINDINGS form fails with cause :NO-BINDINGS.)

The special reader syntax #?VAR accesses the logical binding of ?VAR. The :INHERIT-BINDINGS keyword causes the bindings in a WITH-QUERY-BINDINGS form to be constrained by any bindings that were established by an enclosing WITH-QUERY-BINDINGS form.

For example, the following code will try all known widgets until it finds one that it can operate successfully:

```
(with-query-bindings
 '(is-a ?widget widget)
 (with-recovery-procedures
  (:general-failure
   (next-bindings))
  (operate-widget #?widget)))
```

3. THE PROBLEM

The subset of ESL described in the previous section provides mechanisms for synchronizing tasks and recovering from unexpected contingencies, but it does not relieve the programmer of the burden of insuring that parallel tasks do not interfere with one another. For example, there is nothing to prevent two tasks from simultaneously trying to achieve mutually contradictory conditions, such as having a device be simultaneously on and off.

Managing such interactions in general is an extremely difficult problem. Simply determining that two conditions are mutually contradictory is itself an intractable problem, since conditions in ESL can be arbitrary predicates, and thus involve arbitrary computations. Even if contradictory goals can be detected, deciding how to deal with conflicts is in general intractable because it can require planning and scheduling.

One canonical example of the problems involved in designing a run-time resource management is the following: a domestic robot is frying chicken when a child has an accident and has to be driven to the hospital. The desired behavior is that the robot turns off the stove before leaving the house in order to avoid burning the house down. However, turning off the stove in the middle of cooking is not normally part of the chicken-frying task, nor is it normally part of taking a child to the hospital. It is the result of an interaction between the two tasks when they are carried out in unison. Specifically, it is the result of an interaction having to do with a conflict over a resource, namely, the robot's location.

A more realistic example in spacecraft is the following: a particular valve must be actuated in order to initiate an engine burn. The valve has a large transient power draw, and would cause a bus trip if performed while the spacecraft has a normal complement of devices powered on. The "correct" response is to power off one or more devices (turn off the stove), then actuate the valve (take the kids to the hospital), and then turn the devices back on (finish frying the chicken).

The general structure of this problem is that a high priority task is in conflict with a low-priority task over a shared resource. One possible solution is to provide a mechanism by which a high-priority task can simply abort a low-priority task when a resource conflict arises. The low-priority task can use a cleanup procedure to perform the load shedding operation (or turn off the stove) before it terminates. This works in many cases, but it is not a general solution because the cleanup procedure itself may conflict with some other task or cause a deadline violation. For example, consider a different kind of contingency in the chicken-frying scenario: suppose a robber comes in to the kitchen waving a gun. In such a situation it might not be appropriate to waste time turning off the stove. Alternatively, it might be acceptable for the robot to leave a burning stove unattended for short periods of time. A mechanism for handling such a situation must at the very least have a vocabulary for one task to communicate to another its intentions to usurp a resource for a limited time, and possibly with many other parameters (including probabilistic parameters) as well. Simply designing a *representation* for the general case is a significant challenge, let alone a computational mechanism for actually making runtime decisions based on the information provided.

4. PROPERTY LOCKS

ESL provides a mechanism for solving a very constrained version of the problem of controlling inter-task conflicts through a mechanism called a *property lock*.

A *property* is a logical assertion whose final value is guaranteed unique. For example, POWER-STATE is a property, since it can be either ON or OFF, but not both at once. (An example of a logical assertion that is not a property is CONNECTED-TO, since a thing can be connected to any number of other things.) ESL provides a mechanism for managing inter-task interactions that can be expressed as properties. This provides a simple heuristic for determining when two tasks conflict: if two tasks attempt to achieve properties that are identical but for their final values then a conflict exists.

A property lock is a data structure that signals a task's intention to make a property take on a particular value. Property locks are used to coordinate tasks so that they do not try to achieve different values for a single property at the same time.

Property locks work as follows: A task wanting a property P to have a certain value V expresses that desire by SUBSCRIBING to a property lock for P. The subscription process can have three outcomes:

1. No other task is subscribing to that lock, in which case the subscription is successful, and the task is said to have SNARFED the lock. (To snarf == to successfully subscribe.) This task becomes the OWNER of the lock.

2. Some other task is subscribing to the lock, and the values that the two tasks want the property to have are compatible. In this case the task snarfs the lock but does not become the lock's owner.

3. Some other task is subscribing to the lock and the values are incompatible. In this case the subscription FAILS with cause :PROPERTY-LOCK-UNAVAILABLE. (A special form is provided that causes such failures to be ignored, called WITHOUT-PROPERTY-LOCK-FAILURES.)

The owner of a lock, once it has snarfed the lock, attempts to actually make the property true by calling ACHIEVE on the property. All secondary subscribers wait for the property to be achieved by the owner. If the owner's call to ACHIEVE fails, then all of the lock's subscribers fail with cause :CONDITION-NOT-ACHIEVED.

Once a lock property has been achieved, the lock's subscribers, which were waiting for the owner to achieve the property, continue to run. If the lock's property subsequently becomes false, then the lock property is said to be VIOLATED. (Note: a lock property can only be violated AFTER it is achieved for the first time.) When a lock property is violated then all the lock's subscribers fail with cause :MAINTAINED-PROPERTY-VIOLATION.

Automatic Recoveries

Maintained property violations are detected by a daemon (i.e. a constantly running background process). This daemon will also attempt to restore or *recover* violated properties by calling a user-specified function on the violated property. Normally, this function is simply the ESL ACHIEVE function, but the Remote Agent uses a different mechanism. (See below.)

Because of the existence of automatic recoveries, one possible response for a task that has failed with a :MAINTAINED-PROPERTY-VIOLATION is to simply wait for the recovery daemon to automatically restore the property. A special form, WITH-AUTOMATIC-RECOVERIES is provided that does this.

If the daemon is unable to restore a violated lock's property then the lock's subscribers fail with cause :UNRECOVERABLE-PROPERTY-VIOLATION.

One issue that arises when implementing an automatic recovery feature is that the recovery may be rendered moot if none of the subscribers to a violated property can tolerate temporary violations. If no task actually waits for an automatic recovery of a violated property then the violation will result in all subscribers exiting the dynamic context of the property lock subscription, and the recovery is rendered moot. This problem is solved in ESL by explicitly yielding control to the task scheduler whenever a property violation occurs. This provides all tasks with an

opportunity to decide whether to wait for an automatic recovery or to release their subscription to the violated lock (and possibly other locks as well). Only if subscribers to the lock persist after this process does the automatic recovery daemon attempt to restore the violated property.

The overall control flow of the property lock mechanism is shown graphically in figure 1. There is one additional complication in our architecture that is not shown in the figure, which was alluded to above. The arrow labelled "monitors" represents more than just monitors. There is also a sophisticated model-based reasoning system that deduces device states that are not directly observable [11].

This system is also capable of generating command sequences to restore device states to desired values. The maintain-properties daemon uses this facility to generate plans for restoring violated properties. So in the Remote Agent two different mechanisms are used to achieve desired states. The ESL ACHIEVE construct is used to achieve a property when a lock is first snarfed, and the model-based recovery generator (external to ESL) is used to restore states in the event of unexpected contingencies. There is no particular advantage to doing things this way. We did this in order to exercise all of the capabilities of the system.

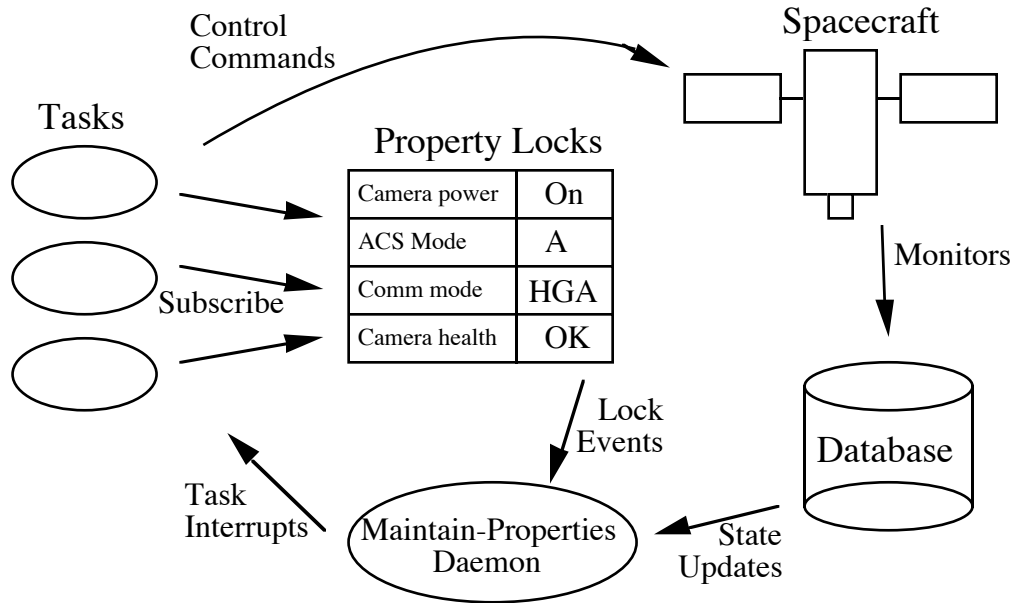


Figure 1: Graphical illustration of the control flow for the property lock mechanism.

5. APPLICATIONS

The property lock mechanism has been used in the DS1 Remote Agent Executive to implement the configuration management subsystem, which is a collection of utilities that embodies knowledge of configuration-management concepts such as redundancy, device mode management, and generic failure recovery. The use of these utilities greatly simplifies the coding of configuration management routines for any spacecraft. In general, all that is required is to describe the configuration of the spacecraft by listing the devices and their interconnections. The system knows about the concepts of power and data busses, single-event upsets, and redundancy.

The system is given information about the spacecraft information through the ESL database. For example, here are all the database entries relating to the x-axis engine gimbal actuator (EGA) at the beginning of a run:

```
(IS-A :EGA_X :EGA_X_FACING)
(TALKS-THRU :EGA_X :GDE)
(POWERED-THRU :EGA_X :NEB1)
(SWITCHED-THRU :EGA_X
 :ACS_GDE_IPS_X_SW1)
(:HEALTH_STATE :EGA_X :OK)
(:POWER_STATE :EGA_X :OFF)
```

We see that EGA_X is an x-facing engine gimbal actuator. It communicates through the gimbal drive electronics (GDE). It is powered through the first non-essential bus (NEB1). Its power is controlled by a switch called ACS_GDE_IPS_X_SW1. It is currently working properly and powered off.

The configuration management system contains routines for configuring devices into high-level abstract states. For example, a *ready* state is defined as one where the device is powered on and ready to receive commands. The system tracks the spacecraft configuration information through the

database in order to decide which low-level configuration changes to make in order to bring about such a state.

To bring EGA_X into the abstract ready state the system first attempts to turn the power on. To do this it first queries the database to find out which switch controls the power to EGA_X, and then sends a command to the switch to turn itself on. Faults are handled automatically through a series of recovery procedures. First the system simply tries the switch again. If that fails, it checks for the presence of a redundant switch and tries that. If that fails, it looks to see if the device itself has a redundant backup and tries that.

Once the power is successfully turned on the system then attempts to establish a communications pathway to the device. This involves a similar string of operations to get the required communications devices (in this case the gimbal drive electronics) recursively into a ready state.

The situation is complicated somewhat by the presence in our architecture of a model-based diagnosis system [11] that can deduce, for example, whether failures are transient or permanent. If a fault is diagnosed as a permanent fault then certain recoveries (like trying again) are bypassed because it is known they will not work. The model-based diagnosis system is also capable of generating sequences of actions to recover from certain fault situations. This capability is used by the recovery daemon.

The property-lock mechanism is used by the configuration management system to insure that mutually-conflicting configurations are not attempted simultaneously. All abstract configurations are resolved into locks on the component states. So, for example, if a task requests that EGA_X be made ready, another task will not be able to obtain a lock on (power-state ega_x off). Of course, nothing actually prevents a task from turning EGA_X off; the property lock mechanism requires cooperation among tasks to be effective. If this happens, the maintain-properties daemon will see this as a fault and attempt to restore the required state by turning the power back on.

6. SUMMARY

An executive for an autonomous agent faces the problem of how to manage interactions among parallel tasks that attempt to achieve mutually contradictory conditions. In its full generality this is an intractable problem. We have presented one limited solution based on the idea of memory properties, logical assertions with unique final values. Tasks coordinate their effects by declaring their intentions to make a property assume a certain value through a data structure called a property lock. This mechanism has been implemented in a system called ESL, and is being used in a control system for an autonomous spacecraft.

We note that property locks tend to have the following form:

```
(property-name object-name value)
```

This structure suggests an alternate implementation in terms of object-oriented design. Instead of a logical database,

system state can be stored in objects whose slots have the names of the properties of interest. So, for example, a device object would have a slot for its power state, its health state, its communications connections, its power connections, and so on. This implementation could be more efficient than the logical-database currently used in ESL, but it compromises some functionality. For example, the logical database makes it easy to find a healthy device of a certain type simply by doing a logical query of the form:

```
(and (is-a ?x ?type) (health-state ?x :ok))
```

To obtain this functionality in an object-oriented database in general is very difficult. Whether it is worth putting forth this effort, or compromising on generality, to obtain greater efficiency for the more common cases remains to be seen.

REFERENCES

- [1] R. Peter Bonasso, et al. "Experiences with an Architecture for Intelligent Reactive Agents," *Journal of Experimental and Theoretical AI*, to appear.
- [2] R. James Firby. *Adaptive Execution in Dynamic Domains*, Ph.D. thesis, Yale University Department of Computer Science, 1989.
- [3] Erann Gat. "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents" *Proceedings of the 1997 IEEE Aerospace Conference*.
- [4] Erann Gat. "Integrating Reaction and Planning in a Heterogeneous Asynchronous Architecture for Controlling Real World Mobile Robots," *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, 1992.
- [5] Erann Gat. "News From the Trenches: An Overview of Unmanned Spacecraft for AI Researchers," Presented at the 1996 AAAI Spring Symposium on Planning with Incomplete Information.
- [6] Erann Gat. "The ESL User's Guide", unpublished. <http://www-aig.jpl.nasa.gov/home/gat/esl.html>
- [7] Damian Lyons. "Representing and Analyzing action plans as networks of concurrent processes," *IEEE Transactions on Robotics and Automation*, 9(3), June 1993.
- [8] Drew McDermott. "A Reactive Plan Language," Technical Report 864, Yale University Department of Computer Science.
- [9] Barney Pell, et al. "An Autonomous Spacecraft Agent Prototype." *Autonomous Robots*, to appear.
- [10] M. J. Schoppers. "Universal Plans for Reactive Robots in Unpredictable Domains," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1987.

- [11] Brian Williams and Pandurang Nayak. "A Model-Based Approach to Reactive Self-Configuring Systems." Proceedings of AAAI96.



Dr. Erann Gat is a senior member of the technical staff at the Jet Propulsion Laboratory, California Institute of Technology, where he has been working on autonomous control architectures since 1988. In 1991 Dr. Gat developed the ATLANTIS control architecture, one of the first integrations of deliberative and reactive components to be

demonstrated on a real robot. ATLANTIS was used as the basis for a robot called Alfred which won the 1993 AAAI mobile robot contest. Dr. Gat was also the principal architect of the control software for Rocky III and Rocky IV, the direct predecessors of the Pathfinder Sojourner rover. Dr. Gat escapes the dangers of everyday life in Los Angeles by pursuing safe hobbies like skiing, scuba diving, and flying small single-engine airplanes.



Dr. Barney Pell is a Senior Computer Scientist in the Computational Sciences Division at NASA Ames Research Center. He is one of the architects of the Remote Agent for New Millennium's Deep Space One (DS-1) mission, and leads a team developing the Smart Executive component of the DS-1 Remote Agent. Dr. Pell received a

B.S. degree with distinction in Symbolic Systems at Stanford University. He received a Ph.D. in computer science at Cambridge University, England, where he studied as a Marshall Scholar. His current research interests include spacecraft autonomy, integrated agent architecture, reactive execution systems, collaborative software development, and strategic reasoning. Pell was guest editor for Computational Intelligence Journal in 1996 and has given tutorials on autonomous agents, space robotics, and game-playing.

Acknowledgements— Ron Keesing and Chris Plaunt contributed to the design and implementation of the resource management routines described in this paper. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.