ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents

Erann Gat
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
gat@jpl.nasa.gov

Abstract—ESL (Execution Support Language) [5] is a language for encoding execution knowledge in embedded autonomous agents. It is similar in spirit to RAPs [2] and RPL [7], and RS [6], 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 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, and is being used to build the executive component of a control architecture for an autonomous spacecraft [8].

1. INTRODUCTION

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 [3]. 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 [7], and RS [6], 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, and is being used to build the executive component of a control architecture for an autonomous spacecraft [4,8].

The following sections provide a brief overview of most of the major feature sets in ESL. For a complete (though terse) description of the language see the ESL User's Guide [5].

2. 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 [9] 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.)

Examples

To illustrate the use of ESL's contingency handling constructs, consider a widget that is operated with the primitive call OPERATE-WIDGET. This call fails cognizantly is the widget is broken. We can break the widget by calling BREAK-WIDGET, which takes an optional argument to specify how badly to break the widget. There is also a repair primitive, ATTEMPT-WIDGET-FIX, which will fix the widget if passed an argument that matches the current widget state. The following execution trace illustrates the basic principles of widget physics:

```
? (operate-widget)
OPERATING WIDGET SUCCESSFULLY.
NIL
? (break-widget)
:BROKEN
? (operate-widget)
Failure :WIDGET-BROKEN, no recovery
available.
Aborted
? (attempt-widget-fix :broken)
Widget is fixed.
NIL
? (operate-widget)
OPERATING WIDGET SUCCESSFULLY.
NIL
? (break-widget :severely-broken)
:SEVERELY-BROKEN
? (operate-widget)
```

```
Failure :WIDGET-BROKEN, no recovery
available.
Aborted
? (attempt-widget-fix :broken)
Widget fix didn't work.
NIL
? (operate-widget)
Failure :WIDGET-BROKEN, no recovery
available.
Aborted
? (attempt-widget-fix
       :severely-broken)
Widget is fixed.
NIL
? (operate-widget)
OPERATING WIDGET SUCCESSFULLY.
NIL
?
```

Notice that attempting to operate the widget in a broken state results in a cognizant failure. Now consider the following code:

```
(defun recovery-demo-1 ()
  (with-recovery-procedures
    ( (:widget-broken
        (attempt-widget-fix :broken)
        (retry))
      (:widget-broken
        (attempt-widget-fix
          :severely-broken)
        (retry))
      (:widget-broken :retries 3
        (attempt-widget-fix
           :weird-state)
        (retry)) )
    (operate-widget)))
```

This code provides three different recovery procedures for recovering from widget failures. The operation of this code is illustrated by the following execution trace:

```
? (recovery-demo-1)
; If the widget is OK nothing special
happens
OPERATING WIDGET SUCCESSFULLY.
NIL
? (break-widget)
:BROKEN
? (recovery-demo-1)
; If the widget is broken it now gets
fixed
Failure :WIDGET-BROKEN, recovery
available.  (No retries)
Widget is fixed.
OPERATING WIDGET SUCCESSFULLY.
NIL
? (break-widget :severely-broken)
:SEVERELY-BROKEN
? (recovery-demo-1)
; First attempt to recover from a
simple broken state
Failure :WIDGET-BROKEN, recovery
available.  (No retries)
Widget fix didn't work.
; Now try the second recovery
procedure
Failure :WIDGET-BROKEN, recovery
available.  (No retries)
Widget is fixed.
OPERATING WIDGET SUCCESSFULLY.
NIL
? (break-widget :irrecoverably-broken)
:IRRECOVERABLY-BROKEN
? (recovery-demo-1)
; None of the recovery procedures will
work, but the third
; one gets three retries before giving
up.
Failure :WIDGET-BROKEN, recovery
available.  (No retries)
Widget fix didn't work.
Failure :WIDGET-BROKEN, recovery
available.  (No retries)
Widget fix didn't work.
Failure :WIDGET-BROKEN, recovery
available.  (2 retries)
Widget fix didn't work.
Failure :WIDGET-BROKEN, recovery
available.  (1 retry)
Widget fix didn't work.
Failure :WIDGET-BROKEN, recovery
available.  (No retries)
Widget fix didn't work.
Failure :WIDGET-BROKEN, no recovery
available.
Aborted
?
```

3. 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:

```
(defun widget-ok? ()
  (eq *widget-status* :ok))
```

```
(to-achieve (widget-ok?)
  ( (eq *widget-status* :broken)
    (attempt-widget-fix :broken) )
  ( (eq *widget-status*
        :severely-broken)
    (attempt-widget-fix
        :severely-broken) ))
```

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. In this case there are two methods, one for the broken state and another for the severely broken state. The operation of this code is illustrated in the following execution trace, beginning with an unbroken widget:

```
? (operate-widget)
OPERATING WIDGET SUCCESSFULLY.
NIL
? (achieve (widget-ok?))
(WIDGET-OK?) achieved.  (No action
needed.)
NIL
? (break-widget :broken)
:BROKEN
? (achieve (widget-ok?))
Attempting to achieve (WIDGET-OK?)
Widget is fixed.
(WIDGET-OK?) achieved.
T
? (break-widget
        :irrecoverably-broken)
:IRRECOVERABLY-BROKEN
? (achieve (widget-ok?))
Attempting to achieve (WIDGET-OK?)
Failure :NO-APPLICABLE-METHOD, no
recovery available.
Aborted
?
```

4. 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))
```

5. 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)))
```

7. Property Locks

One of the problems in multi-threaded agent code is unintended interactions that occur through external effects. For example, one thread may turn a device on while another thread is trying to turn it off. In its full generality this is a very complex unsolved problem. ESL provides a mechanism for solving a constrained version 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 tasks'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.

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, so one possible response for a task that has failed with :MAINTAINED-PROPERTY-VIOLATION is to simply wait for the daemon to automagically 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.

7. SUMMARY

This paper has described ESL (Execution Support Language), a language designed for encoding execution knowledge in embedded autonomous agents. ESL consists of several independent sets of features, including constructs for contingency handling, task management, goal achievement, and logical database management. The contingency handling is based on a cognizant-failure model, while task synchronization is based on a first-class event data object.

Unlike its predecessors, RAPs, RPL and RS, ESL targets a more utilitarian point in the design space, aiming to be a useful programming tool first, and a representation for automated reasoning or formal analysis second. It does this by offering a toolkit of loosely coupled, freely composable constructs rather than a constraining framework. In this respect, ESL is similar to TCA [10]. The main difference between ESL and TCA is that TCA is a C subroutine library, whereas ESL includes new control constructs that have no analog in C and thus cannot be duplicated in TCA. For example, TCA cannot abort a task that has gotten into an infinite loop; ESL can. Also, because ESL introduces new syntax, it allows similar functionality to be achieved in fewer lines of code.

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, "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.

[4] 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.

[5] Erann Gat, "The ESL User's Guide", unpublished. http://www-aig.jpl.nasa.gov/home/gat/esl.html

[6] Damian Lyons, "Representing and Analyzing action plans as networks of concurrent processes, " IEEE Transactions on Robotics and Automation, 9(3), June 1993.

[7] Drew McDermott, "A Reactive Plan Language," Technical Report 864, Yale University Department of Computer Science.

[8] Barney Pell, et al. "Plan Execution for Autonomous Spacecraft." Working Notes of the 1997 AAAI Fall Symposium on Plan Execution.

[9] M. J. Schoppers, "Universal Plans for Reactive Robots in Unpredictable Domains," *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1987.

[10] Reid Simmons, "An Architecture for Coordinating Planning, Sensing and Action," Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, 1990.

Dr. Erann Gat is a member of the techincal staff at the Jet Propulsion Laboratory and co-lead of the Smart Executive component of the New Millennium DS1 Remote Agent autonomy system.

Acknowledgements—Barney Pell, Jim Firby and Reid Simmons provided many useful comments on the design of ESL. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.