A Reactive Plan Language

Drew McDermott

YALEU/CSD/RR #864
August, 1991

This work was supported by the Defense Advanced Research Projects Agency, under contract number DAAA15-87-K-0001, administered by the Ballistic Research Laboratory; and contract number N00014-91-J-1577, administered by the Office of Naval Research.

A Reactive Plan Language

Drew McDermott

1 Overview

RPL (Reactive Plan Language)[1] belongs to the family of notations for writing reactive plans for agents (e.g., robots) (Davis 1984, Ingrand and Georgeff 1990, Lyons 1990a,b, Gat 1991). Its immediate ancestor is Firby's (1987, 1989) RAP notation. Many of Firby's concepts have been carried over, but there are some differences:

(1) The syntax is more "recursive," more in the style of Lisp
(2) More high-level concepts (interrupts, monitors) have been made into explicit constructs.
(3) The interpreter does not attempt to maintain a world model that tracks the situation outside the robot.

RPL is being applied to high-level robot planning, and to representing advisory plans for emergency situations (McDermott et al. 1991).

1.1 Sensor-Guided Plans

A RPL plan looks like a Lisp program. (Indeed, I will use the terms "plan" and "program" interchangeably.) A plan consists of a set of procedure calls, of the form `(`*f* *–args–*`)`, glued together with syntactic constructs like `(SEQ` *–steps–*`)`, `(IF` *condition* act_{true} act_{false}`)`, and the like. Variable bindings in RPL are lexically scoped. But Lisp global variables can be accessed, too. There are special facilities for binding variables to the tasks created by the plan itself.

Any Lisp procedure can be called from a RPL plan. In addition, RPL procedures can defined using

`(DEF-INTERP-PROC` *name* `(`*–formals–*`)` *–body–*`)`

There are no local procedures (i.e., no `LABELS`).

RPL looks so much like Lisp that many Lisp programs are valid RPL plans. However, the intent is not to encourage users to think of plans as typical computer programs, whose operation hinges on the maintenance of complex datastructures. RPL plans are intended to spell out how the behavior of an agent is to be driven by events around it, and to do so in a notation so transparent that a planner can reason about how well the plans will work. A key mechanism to achieve these aims is the idea of a *fluent*, or time-varying quantity. Of course, all program variables are time-varying quantities, but in plans we want behavior to be governed by the temporal changes. For example, in RPL we can say `(FILTER` *c e*`)` to mean, "Execute *e* while the fluent

[1] Thanks to Mark Drummond and Sean Engelson for comments on previous drafts of this paper. The work reported herein was supported by the Defense Advanced Projects Agency, under BRL contract DAA15-87-K-0001, and ONR contract N00014-91-J-1577.

c remains true." If *c* becomes false, the execution of *e* is aborted. Fluents can be defined in terms of other fluents. For example,

```
(FILTER (AND C1 (> I 5)) (CARRY-OUT A))
```

does (CARRY-OUT A) only while C1 remains true and I remains greater than 5. Here C1 is a Boolean-valued fluent, and I is a numerical-valued one.

Fluents can be set by sensors, thus allowing immediate sensory control of actions. Many plans could consist only of batteries of parallel monitors waiting to react to various conditions:

```
(PAR (WHENEVER  condition_1  reaction_1)
     (WHENEVER  condition_2  reaction_2)
     ...
     (WHENEVER  condition_n  reaction_n))
```

The terms C1, I, and A in the FILTER example are plan variables, which behave exactly as in standard Lisp-like languages. There are local variables, bound with LET, and global variables, which it is convenient to implement with ordinary Lisp globals. The programmer must distinguish between variables bound to fluents and those bound to more mundane values. In a plan beginning:

```
(LET ((A (CREATE-FLUENT 'A 5))
      (B 6))
   ...)
```

one can write both (+ A 10) and (+ B 10), but "+" has been defined so that the former expression yields a new fluent, and the latter a number. To get the current value of A, use the expression (FLUENT-VALUE A), as in (+ (FLUENT-VALUE A) 10).

1.2 The Task Network

As a RPL plan is executed, a task network is constructed. The task network corresponds to the stack in a standard programming language. However, an important difference is that the "stack frames" can come into being before the interpreter reaches them, and the planner and interpreter can refer to them in advance. A task is an action (token) that the planner has carried out or might try to carry out. A task has two kinds of subtask (McDermott 1985): syntactic subtasks and reducing subtasks. A *syntactic subtask* of a task T is one that is generated from a piece of the text of the action of T; a *reducing subtask* of a task T is one that purports to be an effective way of carrying out T.

For example, if the action of T is (LOOP (A) (B)), then one of its syntactic subtasks might be "the second step in the third iteration of T," i.e., the third execution of (B). A task with an open-ended set of syntactic subtasks is said to be *iterative*. Obviously, only a finite number of subtasks can actually be executed (or even committed to).

Syntactic subtasks are accessible using plan variables, which are bound using the `:TAG` construct. In this plan:

```
(SEQ (A) (:TAG STEP2 (B)) (C))
```

the second step can be referred to using the variable `STEP2`. One use of `:TAG` is in the `PARTIAL-ORDER` construct, which expects explicit constraints among steps:

```
(PARTIAL-ORDER ((:TAG STEP1 (A))
                (:TAG STEP2 (B))
                (:TAG STEP3 (C)))
   (:ORDER STEP1 STEP3)
   (:ORDER STEP2 STEP3))
```

Each **STEP***i* is a plan variable, bound to a particular task, that is, a particular occurrence of an `(A)`, `(B)`, or `(C)` action. The binding is located in the environment belonging to the highest supertask for which there is a single occurrence of the action. There will be multiple occurrences if the `PARTIAL-ORDER` occurs inside an iterative context, such as a `LOOP` or `WHENEVER`. In such a case, the tag binding is located in the task for each iteration, and is rebound each time the `LOOP` or `WHENEVER` goes around again.

Transformations on plans need to be able to refer to arbitrary subtasks, so we provide a general mechanism for that job. Suppose T is an expression whose value is a `LOOP` task. We stipulate that (`SUB ITER 3` T) refers to the task corresponding to the third iteration of the loop. (Cf. McDermott 1985.) Similarly, for each construct in the language, we provide a notation for referring to the syntactic subtasks it gives rise to. (See Sect. 4.4.) Most of the time we can avoid constructing lengthy terms to refer to sub-sub-. . .-subtasks by using the expression (`TAGGED` t T) to refer to the subtask of T with tag t. So the following is legal:

```
(PARTIAL-ORDER
    ((:TAG DOIT (FOO))
     (FILTER CC (:TAG LL (LOOP (A) (:TAG SS (B)) (C)))))
    (:ORDER (TAGGED SS (SUB ITER 3 LL))
            DOIT))
```

This plan says to do `(A)`, `(B)`, and `(C)` repeatedly (until `CC` becomes false), and also to do `(FOO)` at some point after the `(B)` step of iteration 3 of the loop.

A new task is created for every "step" of a plan. To avoid generating a huge pile of tasks, we distinguish between *steps* and *expressions*. The latter are pieces of the plan that are *evaluated* rather than *executed*. For example, in (`IF` e a b), the expression e is evaluated, resulting in the usual choice of a or b. A task with an `IF` action has two subtasks, one for the true arm and one for the false arm. There is no subtask for the test, e. It's important that expressions be Lisp forms that have no side effects, that is, Lisp forms that can be evaluated whenever their arguments are known, in case some planning algorithm needs to know what their values will be.

Suppose a task T's action is to call a RPL-procedure. In that case, it will have a single subtask, whose action is the body of the procedure. This subtask is not really syntactic, but we use the **SUB** notation to refer to it all the same, as **(SUB PROC-BODY** T**)**.

There is a similar trick for "syntactifying" reducing subtasks. Suppose that the planner has been given a plan of the form **(SEQ A1 A2 (PAR A3 A4 A5))** to improve, and it has determined that the third and fifth step can be performed more efficiently by doing action **B**. It can replace the text of the plan with

```
(REDUCTION (SEQ A1 A2
                (PAR (:TAG R1 A3)
                     A4
                     (:TAG R2 A5)))
           (R1 R2)
           B
           MY-TAG)
```

This plan is executed just as the original one was, but the execution of **A3** and **A5** is short-circuited, and **B** is done in their place. The task for **B** appears as a subtask of both task **R1** and task **R2**. (There is no other way for a task to have more than one supertask.) The task with action **B** is not a syntactic subtask of **R1** or **R2**, but we can pretend it is, by referring to it as **(SUB REDUCTION MY-TAG R1)** or as **(SUB REDUCTION MY-TAG R2)**.

When a task is first created, it has status *created*. It can be created arbitrarily far in advance of when it is executed. When it is ready to be executed, that is, when no further condition needs to be satisfied before execution can begin, it gets status *enabled*. When execution begins, its status becomes *active*. An active task has three possible fates: it can become *finished*, *failed*, or *evaporated*. The normal end of a task is to "finish," presumably "successfully." A task evaporates if events outside it cause its execution to suspend prematurely. An example is the fate of the task for the body of a **FILTER**, should the filtering fluent become false. The status of a task is stored in a fluent, which can be used to trigger events. See **BEGIN-TASK** and **END-TASK** in Sect. 4.

1.3 Failure

Task failure is a more complex issue. We can distinguish at least three possible concepts of task failure: (1) A task fails if it does not accomplish its purpose; (2) a task fails if it becomes impossible; (3) a task fails if it explicitly gives up. An example of the first is a robot's failed attempt to collect a bunch of ice cubes in a room by leaving them on a table in the room (without noticing that they will melt). An example of the second is failing to collect five blocks before midnight by having collected only four when midnight comes. An example of the third is a robot's failing to retrieve a blue block it left on a table because when it returns it finds two similar-looking blue blocks there.

As far as the RPL interpreter is concerned, the only type of failure it can care about is (3). There is a primitive **FAIL** that never finishes successfully, and causes all its supertasks to fail as well, unless the failure is "caught" by a construct like **TRY-ALL** or **TRY-IN-ORDER** that expresses alternative strategies. If a failure is caught by a **TOP-LEVEL** task, then it is reported to the planner. The idea is that the current plan is inadequate to deal with the world's behavior, and the planner should step in to generate a better one. Meanwhile, execution of the other **TOP-LEVEL** subtasks proceeds, presumably unaffected by the failure.

Because the reactive interpreter cannot be expected to do deep reasoning about whether it is succeeding, it is up to the planner (and the human plan writer) to try to make failures of types (1) and (2) look like (3) if they are to be recognized. For example, to catch a deadline violation, it suffices to have code that checks the time and **FAIL**s if the deadline has passed.

FAIL computes a *failure description*, or "faildescrip," on the spot, which gets transmitted to supertasks. These are implemented as objects with at least these three slots:

TASK	The task that failed or **#F**[2] if unknown.
ID	A descriptive S-expression that is printed when the task is.
SEVERITY	An integer between 0 and 10 that summarizes how intractable the failure is. A value of 0 means that the failure is trivial, caused for some backtracking purpose within the plan. A value of 10 means that the failure is so severe that the plan containing it should not be attempted again.

Faildescrips are implemented as Common Lisp Object System (Keene 1989) classes, so more specific classes of failure can be constructed that include more slots. Such specific information is useful in allowing a planner to choose a way to repair a failed plan. The only such class used by the RPL intepreter is the class **composite-failure**, produced by constructs like **TRY-ALL** when every one of its subtasks fails. It has one extra slot, **COMPONENTS**, which is a list of all the faildescrips for its subtasks. See (McDermott 1991) for more information.

1.4 Concurrency

The partially ordered plan is a commonplace in classical AI planning. However, all that is usually meant by the partial order is that any compatible total order is allowed. With a real agent operating in a real world, we have to allow for true concurrency. That's because all actions take some time, and the agent may want to engage in other activities while waiting for an action to finish. We control the concurrency using a type of semaphore called a *valve*. Whenever two tasks should not execute simultaneously, we arrange that they compete for a valve. The loser waits for the winner to finish and release it.

The entity that requests and waits for a valve is called a *process*. In a sense, every task is a process, but it is convenient to single out a larger unit on which we confer the official process label. The construct (**PROCESS** *p* *-body-*) binds variable *p* to a new process corresponding to the execution of *body*. Within this task, a valve can be requested by executing (**VALVE-REQUEST** *p* *v* [*f*]), where *p* evaluates to a process, *v* to a valve, and *f* to an optional fluent that will be explained later. If another process owns *v*, then the

[2] I use **#T** and **#F** for Boolean true and false.

process p waits until the valve is available before proceeding. Valves become available upon execution of (`VALVE-RELEASE` p v), which occurs implicitly when a process finishes.

RPL's process structure is somewhat unusual. Processes are essentially just parts of the task network, so they have the same hierarchical structure as tasks. When one process $P1$ creates another, it does so simply by having a *piece* of itself be declared a new process $P2$. Hence $P1$ remains active while $P2$ is running. The result is that any given task is actually part of a "stack" of active processes. (There is always at least one, because the top task in the network is given a process when it starts.) The *innermost* process is the one nearest to it in the stack, that is, the one most recently created. The current task may own several valves in the name of various of its processes.

Valves are created using (`CREATE-VALVE` *name* *b*), where *b* is a boolean determining whether the valve is *preemptible*, that is, whether one process can take it away from another. Actually, all valves are preemptible by more urgent processes (those with lower priority numbers; see Section 1.5). In addition, the interpreter will force a preemption when it detects a blockage cycle among processes (to be defined shortly) that can be eliminated by transferring ownership of a valve from one process to another. A process can be notified of such an event by using the optional fluent argument to `VALVE-REQUEST`, which is set to `#T` whenever the process gains control of the valve, and set to `#F` whenever it loses it. (This protocol is observed until the `VALVE-RELEASE` is executed that terminates this `VALVE-REQUEST`.)

A process is *blocked* when none of its active tasks can actually proceed. At any instant, there is a set of *threads of control* for a process, each corresponding to an attempt to make progress on a subtask of that process. The interpreter keeps a queue of enabled threads that are ready to run.[3] Other threads are located in various other places. For example, when a plan executes (`WAIT-FOR` *fluent*), and the *fluent* has value `#F`, the plan suspends, and a thread to resume it is queued up on the fluent, waiting for it to become true. Similarly, (`WAIT-TIME` *interval*) causes a thread to be placed on a queue of threads waiting for time to pass. (`VALVE-REQUEST` *p* *valve*), when the *valve* is in use, causes a thread to be put on a queue associated with the valve. To avoid losing track of what a process is waiting for, the interpreter keeps track of all the threads it currently has queued; these are called *blockages*. The process is blocked when all its threads are located in blockages.

An important class of fluents are those that refer to events within the interpreter itself. (`BEGIN-TASK` t) becomes true when t becomes active. (`END-TASK` t) becomes true whenever t comes to an end (by finishing, failing, or evaporating). Constraints on the ordering of plan steps are realized by causing one task to wait for another to begin or end.

[3] The details are, as usual, more complex. There are actually two queues, the queue `ENABLED*` of enabled threads, and a queue `UNBLOCKED*` containing threads that need more careful analysis before they can be declared enabled.

even when the policy is blocked by using the construct `(WITH-TASK-BLOCKED` *t -body-*`)`. This construct prevents any subtask of *t* from running until the *body* is finished. Typically *t* will evaluate to the primary task or one of its crucial subtasks.

`WITH-POLICY` installs a policy that lasts for exactly the duration of another task. If you want to make the scope narrower, you can use the policy creator

`(SCOPE` *b e policy*`)`

where *b* and *e* are fluents. The resulting policy is equivalent to *policy*, but lasts only from the moment when *b* becomes true to the moment when *e* becomes true. Two useful fluent creators in this context are `(BEGIN-TASK` *t*`)` and `(END-TASK` *t*`)`.

Valves, policies, `WAIT-FOR` and `WITH-TASK-BLOCKED` can be used to make one process wait for another that it could interfere with if allowed to run. There is also a global priority mechanism that allows more urgent processes to interrupt less urgent ones, preempting valves if necessary. Executing `(PRIORITY` *x -body-*`)` causes *body* to be executed with the priority *x*. So far I have had less use for this construct than I expected; presumably it would be valuable in implementing plans for dealing with emergencies.

1.6 Protections and Violations

A *protected state* is a state of affairs that an agent needs to keep true over some interval (Sussman 1975). Classical planning theory is focused largely on the issue of preventing potential *protection violations*, that is, events that cause protected states to become false. As Firby points out (Firby 1989), a reactive plan interpreter must have a more complex view of protection. If a protection is viewed as a run-time plan annotation, then it must specify code to

1 Check periodically whether the protected state is still true
2 Take appropriate action when it becomes false.

RPL contains the `PROTECTION` construct to make this job easier (and to interface to the planner):

`(PROTECTION [:RIGID | :HARD | :SOFT]` *prop f repair*`)`

Here *f* is the fluent being protected. If it becomes false, then *repair* is executed. If it's still false, then a failure occurs.[6]

The *prop* argument to `PROTECTION` is an S-expression that summarizes the "content" of the proposition being protected. This S-expression goes into a global list of active protections, so that other plans can see what commitments they have to respect. The Lisp procedure `(MATCH-PROTECTIONS` *pattern*`)` searches this list for protections with *props* matching *pattern*. It returns a list of association lists, one for each matching *prop*. Each association list specifies a binding for the variables in *pattern* (indicated with a "?"). For example, if a `PROTECTION` with *prop* `(HOLDING !:DESIG20)` is active, then `(MATCH-PROTECTIONS '(HOLDING ?X))` will return the list `(((X !:DESIG20)))`.

[6] It should be of type `protection-violation`, but currently it's a generic failure.

There are two ways, therefore, that one process P_1 can wait for another P_2. P_2 might own a valve that P_1 has requested; or some task in P_1 might be waiting for the beginning or end of some task in P_2.[4] A *blockage cycle* exists among a sequence of processes if each is blocked, and each is waiting for the next in one of these two ways. The interpreter checks for such cycles whenever a process becomes blocked, by tracing blockages. If it finds a cycle, and the cycle involves a preemptible valve, then the valve is given to the process in the cycle that requested it.

It is not always proper for a process to relinquish control when it suspends waiting for some event. For example, a low-level motor controller might have to wait a millisecond for an acknowledgement from the outside world, and we don't want its process to become blocked during that millisecond. All RPL constructs that wait can take an explicit `:KEEP-CONTROL` flag to indicate that during the wait the process will not be pre-empted in order to break a blockage cycle. See Section 4.2.

1.5 Policies and Priorities

Many plans are executed solely to constrain the execution of other plans. Such a "secondary" plan is called a *policy* (McDermott 1978). For example, a robot might go from one place to another, but adopt the policy that if its gripper should ever become empty, that it must stop and pick up the object it dropped. We indicate such a control structure by writing `(WITH-POLICY` s a`)`, where s is a secondary action (e.g., "whenever the gripper becomes empty, regrasp the object"), and a is the primary action it constrains (e.g., "go to the destination").

Policies are constraints, but we model them as ordinary plans, rather than trying to provide a separate constraint language. The reason is simple: the same sorts of construct appear in policies as elsewhere.[5] In a `WHENEVER` such as that alluded to in the previous paragraph, we may well want to include some sequentiality once the triggering event has been detected. We adopt the convention that the constraint is declared violated if the policy should `FAIL`, in which case the whole `WITH-POLICY` task fails. If the primary action succeeds, the policy evaporates and the `WITH-POLICY` task succeeds. Normally it is not necessary for policy tasks to finish, and it is natural to think of a `WHENEVER` as lasting forever unless beset by evaporation. But we take it to be harmless for a policy to finish; the result is that the constraint just goes away.

There is a further necessary stipulation. For a policy to do its job of constraining the execution of its primary task, it must be the case that the primary can never run unless the policy is blocked. Policies must be written so that they normally *are* blocked. When they wake up, they are guaranteed that their primaries remain blocked while they take action in response to events. A policy can prevent its primary from running

[4] There are other ways, too, of course, such as P_1 waiting for an event in the world that, as it happens, only P_2 can cause, but such dependencies are harder to detect.

[5] Put another way: Plans already *are* constraints — constraints on the course of history (McDermott 1985).

Finally, the flag `:RIGID`, `:HARD`, or `:SOFT` is a note to the planner telling it how many violations of this protection we expect. `:RIGID` means that any violation would be bad; repairs are expected to be expensive or impossible; the planner should work hard to forestall violations. `:HARD` means that the planner should try to forestall a violation, but the interpreter should be able to cope with one. `:SOFT` means that violations are routine, and the planner should treat projected violations as normal events. (See Section 1.8.)

1.7 Perception

Suppose a RPL plan is to guide a robot in its manipulation of an object outside itself. If the object is permanently connected to some sort of input/output port, then we can manipulate the object by routines that send signals to, and receive signals from, that port. (See Section 2.4.) But if the object is only intermittently encountered by the agent, then the agent has the problem of recognizing it on each encounter. Solving this problem will require a different sensory process for each application. My topic here is how to connect such solutions to RPL plans.

The standard scenario goes like this: The robot scans its environment with its sensors, and picks up signals indicating the presence of some objects of interest. The signals tell the robot two sorts of things: what the objects are like, and where they are in robot-centered coordinates. Now suppose the robot is looking for a particular object, fitting a certain description, and expected to be in this vicinity. What it must do is check whether there is a single candidate among the recently scanned objects that matches the description given. If so, it can assume that the sought object and the present object are identical.

I have provided the following simple support for this idea. There is a datatype *desig* (for "designator") that represents a perceived or expected object. A desig is basically just a property list and a printed representation. Plans can pass desigs to subroutines to tell them what objects to manipulate. If a desig has the right kind of information on its property list, the manipulation will suceed. E.g., a plan to `PICKUP` a desig must move the gripper to the coordinates stored on the desig's property list, and then close the gripper.

If a desig does not contain the right kind of information, then a plan can invoke the strategy described above for acquiring it. To wit: Scan the world for an object resembling the sought object. Arrange that sensory operations return desigs representing the objects perceived. If a suitable candidate is found, apply the operation `EQUATE` to the found desig and the expected desig. (`EQUATE` *new old*) links the two given desigs. It declares *new* to be the "latest version" of *old*. Subsequent references to the property list of either desig will start with the latest version and work backward chronologically until an entry is found. Hence if the current pose of the object is stored on the new desig, that information will be attached to both the old and new desigs.

This idea is due to Jim Firby. Consult (Firby 1989) for a more elaborate development.

1.8 Plan Projection and Transformation

A planner needs to be able to reason about plans in addition to executing them. This topic is treated at greater length in (McDermott 1991), but a few words are appropriate here.

When a planner believes it has a good version of a plan, it can test that hypothesis by *projecting* the plan (Wilensky 1983, Simmons 1988,Hanks 1990), that is, by "mentally executing" it to see if it succeeds in achieving its goals. RPL supports this idea by being runnable in either *real mode* or *projection mode*. In the latter mode, no actions are actually carried out. Instead, as simulated time passes, a timeline is built. When projection is complete, a *projection* data structure is returned, which includes the timeline, the task network, and bugs encountered, among other things. See Section 2.2 for a few more details.

2 Implementation

RPL is implemented as a Lisp program of about 6000 lines of code, on top of the Nisp (McDermott 1988) macro package.

2.1 Internal Code Representation

When a RPL procedure is defined, its body is transformed into an internal form called "rpl-code." This is a tree of objects that correspond to the task network the procedure can generate, except that it is static.

As discussed in Section 1.2, nonprimitive tasks give rise to subtasks whose names depend on how their actions were derived from the action of the supertask. For example, if task t has action (SEQ a_1 a_2 ... a_n), then it has n subtasks, the i'th having action a_i and name (SUB STEP i t). The list (STEP i) is called the *name prefix* of the subtask.

In rpl-code trees, the same name-prefix system is used, except that even iterative constructs like LOOP have only a finite number of subcodes. The code for (LOOP *-body-*) will have a single subcode for the *body*. This subcode will have the name prefix (ITER *), which will be used to generate subtasks with name prefixes (ITER 1), (ITER 2), etc. The interpreter does not have to reparse each code action, because the subcode structure already captures how the task is to be broken into subtasks. The code is cleaned up in various other ways as well, most notably:

1 All macros are expanded away.
2 Tags are collected and their definitions stored in "tag tables" at the appropriate levels in the rpl-code tree. (See below.)

To refer to a chain of SUB relationships, it is often convenient to write (PATH-SUB *path* t), where *path* is a list of name prefixes, as in (PATH-SUB ((STEP 3) (ITER 1)) TASK101), which abbreviates (SUB STEP 3 (SUB ITER 1 TASK101)). (The name prefixes are (STEP 3) and (ITER 1).) Normally, long chains of name prefixes are a dangerous way to refer to a particular task, since transforming the plan will change what the chain refers to. Tags enable you to skip over segments of such a chain. (TAGGED l t) finds the subtask of t tagged with label l. Tags can be used in paths, by treating (TAGGED l) as a name prefix.

SUB and PATH-SUB are Lisp macros that do not evaluate their first arguments. They are most useful as explicit parts of plans. When writing a Lisp program to manipulate a plan, the procedure (PATH-PLACE-SUB *path task*), which evaluates both its arguments, is often more useful. (TASK-NAME-PATH T) takes a task argument and returns the name-prefix path for it, in order from lowest to highest. This is often more useful than a pointer to the task itself, because it remains invariant from one execution of the plan to the next. TASK-NAME-PATH returns paths containing TAGGED wherever it can.

However, there are occasions when you need to refer to subtasks by explicit name prefix. The standard examples are the use of (ITER i) to refer to the i'th subtask of a LOOP (and similarly for other iterative constructs); and the use of (PROC-BODY) to refer to the body of a called procedure. See Section 4.4 for a table of all subtask name prefixes.

Tags are implemented by giving each rpl-code object a TAGTAB slot, which maps tags bound in it to task-name paths. When an expression of the form (TAGGED ...) is evaluated, the path is followed and the task at the end of it is returned, after being created if it does not already exist.

2.2 The Controller

RPL is designed to be a language for a programmable robot controller. Currently the controller is run as a Lisp process, although in the long run it would be desirable to compile it to run on a lower-level dedicated operating system (Stewart et al. 1990, Hendler and Agrawala 1990).

A RPL plan is run by creating a new top-level task and putting a thread of control to interpret it on the agenda managed by the control process. The Lisp procedure (CONTROL-INTERP *plan*) does all this for you (starting the process if it is not currently active, and evaporating any previously sent plan). During debugging, it is sometimes more convenient to execute (INTERP-RUN *plan*) to start RPL's controller and let it work on *plan*. However, there are cases where INTERP-RUN returns before everything it started has concluded. Calling (RUN) will let the agenda manager run a little more, emptying out queues.

There is a similar procedure (PROJECT-RUN *plan*) that runs the interpreter in projection mode. (Section 1.8) It returns a projection, a data structure including a timeline, a task network, and a record of bugs found. The user should never call PROJECT-RUN directly, because it depends on a few global variable bindings that I'm not telling you about. Instead, call (PROJECT *plan*), which returns a "lazy list" (McDermott 1988) of projections, each obtained by calling PROJECT-RUN. A planner can ask for as many elements of this list as it has time to contemplate, and the projector will keep generating them, in Monte Carlo fashion. Of course, when you call PROJECT, you have to have provided mechanisms for projecting low-level actions. That is, you have to tell it how to project an arm movement without actually moving the robot's arm. See McDermott 1991 for that and other details.

The controller's behavior is influenced by the settings of the following dynamic Lisp variables (most of which are *not* user-settable):

CONTROL-PROCESS*	The Lisp process running the controller.
PROJECTING*	Normally **#F**. Bound to **#T** when the interpreter is running in projection mode. See Section 1.8.
SUBPROCESSES-BLOCK*	If **#T** (the default), then when a subprocess takes a valve from a superprocess, the superprocess can no longer run. If **#F**, then subprocesses do not block their superprocesses. (User-settable)
AGENT-REPORT*	If non-**#F**, must be bound to a function of two arguments, a task and an outcome (either a faildescrip or the value it returned when it finished). The projector binds this to a procedure that adds to its bug list. (User-settable)
SUBTASKS-GC*	If **#T**, then when a task is finished (or evaporated or failed), pointers from it to its subtasks are dropped, allowing their storage to be reclaimed. Bound to **#F** in the projector, so that entire task networks persist until the projection is concluded. (User-settable)
CURRENT-TIME*	Bound to a procedure of no arguments that returns the current time in seconds from some zero point. The system rebinds it inside the projector to a procedure that returns the duration of the timeline so far.
SENSE-QUEUE*	A queue of events set up by the **SENSORY-INPUT** macro. (See Section 2.4.)
PENDING*	
UNBLOCKED*	
ENABLED*	Internal interpreter thread queues. Rebound by projector.
CONTROLLER-RUNS-FOREVER*	If **#T**, the controller sleeps when there is nothing for it to do. This is reasonable behavior when the interpreter is running as a separate process. If it's **#F**, the controller quits when there is nothing for it to do.
NEXT-TIME*	The next time something interesting is scheduled to happen, or **#F** if nothing is. Rebound by the projector, or any other incarnation of the interpreter.

2.3 Lisp and RPL

Any Lisp procedure can be called from RPL. The arguments to a Lisp function must be RPL expressions (not commands).

Some Lisp procedures have different definitions in RPL. E.g., **NOT** checks its argument to see if it's a fluent and if so returns a new fluent. There are analogous definitions for **AND**, **OR**, **>**, **<**, **>=**, **=<**, **EQ**, **+**, and **-**. In each case, the system tries hard to "uniquify" the new fluent. That is, if there is already a fluent produced from `(+ I 1)`, a later evaluation of this expression will return the same fluent again. In general, you should never try to **SETF** the value of a derived fluent, except for the derived fluents `(NOT` f`)` and `(-` f`)`. Setting one of them will set f to the proper inverse value. E.g., setting the value of `(NOT A)` to **#T** will set **A** to **#F**.

These fluent constructors are available from inside Lisp, but are not currently documented.

2.4 Running and Connecting RPL

At some point RPL code must bottom out in links to the outside world, in the form of effector commands and sensor readings. Effector commands should be implemented as *nonblocking* Lisp programs that send signals to output ports. Sensor readings are a little more complex. Suppose an input port can cause a hardware or software interrupt. We need to arrange to bring that to the attention of the CONTROL-PROCESS*. We do that by supplying a Lisp program that handles the low-level interrupt, and executes

(SENSORY-INPUT *-Lisp-code-*).

The effect is to cause *Lisp-code* to be executed inside the control process on its next cycle. The Lisp code will typically set the value of one or more fluents. *IMPORTANT:* Do not use set the value of a fluent outside the control process, except inside the body of a SENSORY-INPUT. If a RPL thread is waiting for a fluent to get a value, setting its value will cause an attempt to run that thread, which only the control process should do. This stricture applies to SET-VALUE, (SETF (FLUENT-VALUE ...) ...), CONCLUDE, and PULSE.

3 Examples

This section presents a few fragments from an evolving system for planning in a simulated robot world. The plans make use of interface code written in Nisp (McDermott 1988), a portable macro package with compile-time typing. Nisp looks like Common Lisp, with a few oddities:

- Variables are declared when bound. A type declaration is a hyphen followed by a type designator. Type designators are either type names (in lower case, by convention), or more complex constructs such as (LST t), which refers to the type of lists of objects of type t. When a list is used as a record, with fixed fields, it is of type (LRCD t_1 t_2 ... t_n), where the t_i are the types of the fields.
- DEFFUNC and DEFPROC are used to define Nisp procedures (without and wide side-effects, respectively). The result type of the defined procedure appears after the procedure's name, preceded by a hyphen.
- If x is declared to be of type t, the expression (!_*slot* x) refers to the given slot of it. The syntax !>*x*.*slot* is also permitted.
- A new object of type t is made by executing (MAKE t *-args-*).
- The macro != is a synonym for SETF, except that any occurrence of *-* on the right-hand side is replaced by a copy of the left-hand side.
- In the examples, occurrences of λ were actually typed in as \\. (λ ...) is an abbreviation for (FUNCTION (LAMBDA ...)).
- The macros <#, <!, <&, and <V are abbreviations of MAPCAR, MAPCAN, EVERY, and SOME, except that their first arguments do not have to have FUNCTION wrapped around them. The macro (<? *pred list*) collects all the elements of *list* that satisfy *pred*. The macro (</ f i l) is an abbreviation for (REDUCE #'f l :INITIAL-VALUE i).
- The booleans are #T and #F. We avoid using NIL, which is usually better written as '() or '#F. The constant T is used only to mean "else" in CONDs
- An expression of type ($\sim$ t) is either of type t or is identically #F.

A few words about the robot's world. The robot moves around on a grid of locations. At each location, it can call (LOOK-FOR-PROPS *property-list*) to run its "vision system." After a delay, the system will report back by pulsing the fluent VISUAL-INPUT* (setting it to #T for an instant), and putting into OB-POSITIONS* a list of local coordinates of objects matching the *property-list*. There are similar actions for inspecting an object at a particular local coordinate, moving a hand to a local coordinate, closing or opening a hand, and so forth.

The robot moves by executing (**OBJ-START-MOVING ROBOT*** v x y), where v is the speed, and $\langle x, y\rangle$ is either $\langle 1,0\rangle$, $\langle 0,1\rangle$, $\langle -1,0\rangle$, or $\langle 0,-1\rangle$. When the move is complete, the fluent **ROBOT-MOVED*** is pulsed.

Here are some plans for finding an object:

```
(DEF-INTERP-PROC LOOK-FOR (PL)
   (LOOK-FOR-PROPS PL)
   (WAIT-FOR VISUAL-INPUT*)
   (SEEN-OB-DESIGS PL)    )


; Returns a list of desigs (Section 1.7), one for every object seen.
(DEFFUNC SEEN-OB-DESIGS - (LST desig) (PL - (LST (LRCD symbol obj)))
   (FOR (I IN OB-POSITIONS*)
        (SAVE (CREATE-DESIG "Perceived object"
                              ;Desig will print like: !:|Perceived object203|
                              (CONS (LIST 'LOC I)
                                    (<# (λ (X) (LIST (CAR X) (CADR X)))
                                        PL))
                              ;Desig property list
              ))))


; Like LOOK-FOR, but return just one
(DEF-INTERP-PROC LOOK-FOR-ONE (PL)
    (LET* ((L (LOOK-FOR PL)))
       (IF (JUST-ONE L) (CAR L) (FAIL)    )))
;(It really should return a faildescrip describing "perceptual confusion.")
```

At a higher level, the robot needs plans for finding and transporting objects. An important construct is (**AT-LOCATION** x y *-body-*), which goes to x, y on the map, then remains there while it carries out *body*.

```
(DEF-INTERP-MACRO (AT-LOCATION ?X ?Y . ?BODY)
   (LET (BODTAG - symbol)
      (COND ((MATCHQ ((:TAG ?BODTAG . ?())) BODY)
             (!= BODY (CAR *-*)))
            (T
             (!= BODTAG (SYMBOL ACTIVITY-THERE (++ ACTIVITY-NO*)))
             (!= BODY '(:TAG ,BODTAG ,(IMPLICIT-SEQ BODY))))    )
;Most of the hair is just to set up a tag for the body, to be used by WITH-TASK-BLOCKED:
      '(LET ((DEST-X ,X) (DEST-Y ,Y))
          (GO DEST-X DEST-Y)
          (WITH-POLICY (LOOP
                          (WAIT-WITH-TIMEOUT ROBOT-MOVED* 60 :ALLOW-INTERRUPTS)
                          (WITH-TASK-BLOCKED ,BODTAG (GO-NEAR DEST-X DEST-Y 2))    )
             ,BODY    ))))
```

The construct `(SYMBOL ACTIVITY-THERE (++ ACTIVITY-NO*))` creates a new symbol, with name `ACTIVITY-THERE`k, where k is obtained by incrementing a global counter `ACTIVITY-NO*`. Half the code of the macro is just to wrap a `:TAG` around the body so it may be blocked by the policy using `WITH-TASK-BLOCKED`.

An example will perhaps be clearer:

```
(AT-LOCATION (+ U 1) (- V 1)
   (ACQUIRE D33)  )
```

expands into

```
(LET ((DEST-X (+ U 1))
      (DEST-Y (- V 1)))
   (GO DEST-X DEST-Y)
   (WITH-POLICY (LOOP
                   (WAIT-WITH-TIMEOUT ROBOT-MOVED* 60 :ALLOW-INTERRUPTS)
                   (WITH-TASK-BLOCKED ACTIVITY-THERE666
                      (GO-NEAR DEST-X DEST-Y 2)))
      (:TAG ACTIVITY-THERE666 (ACQUIRE D33))))
```

`GO` takes the robot to a destination; `GO-NEAR` does, too, but only to within a tolerance. Hence if the `ACQUIRE` does not succeed right away, and starts to move the robot away from ⟨`(+ U 1)`,`(- V 1)`⟩, the policy will bring it back.

`ACQUIRE` takes a desig as argument, and finds it in the world. It searches until it finds an object that resembles the one sought, and equates the desigs for the two. The desig for the object just found will have enough information to enable the robot to manipulate it.

```
(DEF-INTERP-PROC ACQUIRE (OB)
   (LET ((PL (PERCEPTUAL-PROPS OB)))
      (WITH-POLICY
         ; Move through grid systematically, delaying
         ; between moves to allow activity to take place:
         (EXPLORE)
         ;  FIND is like  LOOK-FOR, but it keeps
         ; looking until it finds something:
         (LET* ((DOPPEL (FIND PL)))
            (IF (JUST-ONE DOPPEL)
                (EQUATE (CAR DOPPEL) OB)
                (FAIL))))))

;; Return the visible properties of an object.
(DEFFUNC PERCEPTUAL-PROPS - (LST (LRCD symbol obj)) (OB - desig)
   (<? (λ (P) (MEMQ (CAR P) '(CATEGORY COLOR TEXTURE FINISH)))
       (!_PROPS OB)))
```

The next plan gets an object into a hand:

```
(DEF-INTERP-PROC ACHIEVE-IN-HAND (OB HAND)
; First, check to see if you believe the object is already there:
   (IF (NOT (OBJ-IN-HAND OB HAND))
       (SEQ
        ; If not, find it and pick it up
          (ACQUIRE OB)
          (IF (NOT (EMPTY HAND))
              (UNHAND HAND))
          (PICKUP OB HAND))))
```

Finally I present the plan for transporting an object from one place to another. It's a longer plan, and shows more of the flavor of the language than the smaller examples. The idea is to pick the object up (step `TAKE-IT`), then go to the destination and release it (step `RELEASE-IT`). However, the object might be dropped along the way (possibly because of interrupts to handle other tasks). Should this occur, the robot is to remember where it was when the object was dropped, and go back and get it when it has a chance. Typically its next chance will be when it gets control of the valve `WHEELS*` again. This valve controls which plan gets to steer the robot somewhere, and must be requested by any plan that needs to do that.

```
(DEF-INTERP-PROC TRANSPORT (OB X1 Y1 X2 Y2)
      (LET ((HAND '#F)
            (DROPPED-IT (STATE '(DROPPED ,OB)))
            (DROP-X 0) (DROP-Y 0))
        (WITH-POLICY (SCOPE (END-TASK TAKE-IT)
                            (BEGIN-TASK LET-GO)
                           (WHENEVER (NOT DROPPED-IT)
                              (SEQ (WAIT-FOR (EMPTY HAND))
                                   ; This part is executed outside the
                                   ; WITH-VALVE, so it does not have to
                                   ; wait for WHEELS* to be free
                                   (!= < DROP-X DROP-Y >
                                       (COORDS-HERE))
                                   (CONCLUDE DROPPED-IT))))
              (WITH-VALVE WHEELS*
                 (PLAN ((:TAG CARRIER
                              ; The core of the plan is here:
                              (SEQ (:TAG TAKE-IT
                                         (AT-LOCATION X1 Y1
                                               (SEQ (!= HAND (FREE-HAND))
                                                    (CONCLUDE (IN-USE HAND))
                                                    (ACHIEVE-IN-HAND OB HAND))))
                                   (:TAG RELEASE-IT
                                         (AT-LOCATION X2 Y2
                                               (SEQ
                                                 (:TAG LET-GO (UNHAND HAND))
                                                 (CONCLUDE (NOT (IN-USE HAND))))))))))
(SCOPE (END-TASK TAKE-IT)
       (BEGIN-TASK LET-GO)
      (PROTECTION :HARD
                  '(TAKING ,OB)
        ; What gets protected is the negation of
        ; the DROPPED-IT fluent, set to #F above
        (NOT DROPPED-IT)
        ; Make sure the main plan waits for
        ; the repair to finish:
        (WITH-TASK-BLOCKED CARRIER
          ; You don't have to use the same hand
          ; after dropping the object:
          (!= HAND (FREE-HAND))
          (CONCLUDE (IN-USE HAND))
          (AT-LOCATION DROP-X DROP-Y
             (ACHIEVE-IN-HAND OB HAND))
          ; Make sure the repair resets the
          ; fluent value:
          (CONCLUDE (NOT DROPPED-IT)))))))))
```

4 A RPL Manual

4.1 Macros

RPL macros can be defined using (DEF-INTERP-MACRO *pattern -body-*). The *body* is a piece of *Lisp* code that explains what to transform *pattern* to when it occurs in RPL. The *pattern* should start with an atom, and have variable parts marked with question marks. E.g., here is the definition of TEST:

```
(DEF-INTERP-MACRO (TEST ?TEST ?IFTRUE ?IFFALSE)
    '(LET* ((TEST-OUTCOME ,TEST))
        (IF TEST-OUTCOME ,IFTRUE ,IFFALSE))    )
```

Macros are expanded out of plans when they are handed to the cpu manager. They are expanded out of procedure bodies when the procedures are defined. Hence code walkers should expect never to see them.

4.2 RPL Commands

(!= *v c*): Execute *c*, and set plan variable *v* to the result. To set multiple values, write

(!= < v_1 ... v_n > *c*).

If a fluent occurs on the left-hand side of a !=, then its *value* is set. So if I is bound to a fluent, (!= I (+ I 1)) sets I's value to the value of (+ I 1), which is itself a fluent.

(ESTABLISH-CONTEXT *p m b e*): Do *b*. If fluent *p* becomes false, do *m* to make it true again. If *m* fails or doesn't make *p* true, evaporate *b* and do *e*. If *p* is false initially, do *m* first, and if it fail or doesn't make *p* true, just do *e*.

(FAIL [|:KEEP-PROJECTING] [*fd* | [|:IMPOSSIBLE|:BACKTRACKING]
[| :CLASS fail-class-name] --args--]):

Fail, transmitting a failure description to supertasks. Most of them will just fail themselves and pass the description along. Some will attempt alternative strategies. If a single argument *fd* is provided, it is assumed to be a failure description. Otherwise, a failure description is built with the material provided. If **:CLASS** *c* is present, then a call to **MAKE-INSTANCE** is generated that takes the subsequent *args* as arguments. Otherwise, the *args* are used to make a generic faildescrip. If there are no args at all, a faildescrip with **DESCRIP=GENERIC** is produced.

The three flags **:KEEP-PROJECTING**, **:IMPOSSIBLE**, and **:BACKTRACKING** are handled specially. The last two are turned into **:SEVERITY** arguments, of values 10 and 0 respectively, for the faildescrip builder. The **:KEEP-PROJECTING** argument causes the failure to be handled specially if the RPL interpreter is running in projection mode. In this mode, **AGENT-REPORT*** will be called immediately (see **TOP-LEVEL**, below), but execution will continue. The intent is to report the failure, but see if worse problems will follow, even assuming that this one is repaired.

(FILTER *p -body-***)**: Do *body* but if *p* (a fluent) becomes false, evaporate.

(IF *p* a_1 a_2**)**: Evaluate *p*. If true, do a_1, else a_2. (If *p* evaluates to a fluent, test its value.)

(LET ((x_1 e_1) ... (x_k e_k)) c_1 ... c_n**)**: The e_i are expressions, not commands. They are evaluated, then the x_i are bound to the results. x_i is either a single variable, or a segment of the form < v_1 ... v_{l_i} >, which binds the v_j to multiple values returned as e_i. Once the x_i are bound, the $c_1, \ldots, c_n$ are executed in order as subcommands. (Just as if they appeared in a **SEQ**, with identical name-prefix conventions.)

(LET* ((x_1 c_1) ... (x_k c_k)) c_{k+1} ... c_n**)**: $c_1, \ldots, c_n$ are executed in order as subcommands. (Just as if they appeared in a **SEQ**, with identical name-prefix conventions.) However, the variable bindings during the execution of each c_i are augmented by binding the preceding $x_1, \ldots, x_{i-1}$ appropriately. x_i is either a single variable, or a segment of the form < v_1 ... v_{l_i} >, which binds the v_j to multiple values returned as e_i.

(LOOP *-body-***)**: Do actions in body in sequence, then repeat. The body may contain tests of the form

```
WHILE e
UNTIL e
```

If *e* evaluates to the right boolean value, the loop exits.

The atom **NEXT-ITERATION** in the body indicates that the next iteration should begin at that point, concurrently with the rest of the current iteration. (Cf. Lyons 1990a.) When a **WHILE** or **UNTIL** test causes the loop to end on some iteration, all concurrent iterations evaporate.

(NO-OP): Succeed. That is, do nothing.

(N-TIMES *n -loop-body-***)**: Like **LOOP**, but quits after *n* iterations if no other test has caused it to exit.

(PAR a_1 a_2 ... a_n**)**: Do actions in parallel. Succeed if they all succeed; fail if one fails.

(PARTIAL-ORDER (*-steps-*) *-orderings-*)**:** Do the *steps* in parallel, as in **PAR**, except as constrained by the *orderings*, each of the form (**:ORDER** s_1 s_2). Each s_1 and s_2 must evaluate to a descendant (in the subtask hierarchy) of the **PARTIAL-ORDER** task.

(PERIODICALLY *t* [| **:KEEP-CONTROL** | **:ALLOW-INTERRUPTS**] *a*)**:** Do *a* every *t* seconds. More precisely, every *t* seconds begin a new occurrence of *a*, whether or not the previous one has finished. The **:KEEP-CONTROL/:ALLOW-INTERRUPTS** flag determines whether the thread blocks between occurrences of *a*. (See **WAIT-FOR**.)

(PLAN (*-steps-*) *-constraints-*)**:** A macro that combines **WITH-POLICY**, **PARTIAL-ORDER**, and **REDUCTION**. Each *constraint* is either of the form (**:ORDER** s_1 s_2), or of the form (**:REDUCTION** *reducees r* [*tag*]), or is a command. The first form gives rise to **PARTIAL-ORDER** orderings; the second form gives rise to **REDUCTION**s; and the third gives rise to policies. Except as constrained, the steps are simply executed in parallel.

(PRIORITY *n -body-*)**:** Execute body with priority *n*. Lower numbers are more urgent. "Normal" priority is 10. Top priority is 1.

(PROCESS *v -body-*)**:** Create a new process and bind *v* to it. Execute *body* inside that process, then flush the process and the binding of *v*.

(PROTECTION [**:RIGID** | **:HARD** | **:SOFT**] *prop f repair*)**:** Whenever fluent *f* becomes false, then execute *repair*. If it's still false after that, **FAIL**. The *prop* is an S-expression that is accessible using **MATCH-PROTECTIONS**. The first argument to **PROTECTION** is a flag for the planner. See Section 1.6.

(PURSUE a_1 a_2 ... a_n)**:** Do actions in parallel. Succeed if one succeeds (the others evaporate); fail if one fails.

(REDUCE *cmd method* [*tag*])**:** Do *cmd* by doing *method*. Equivalent to *method*, but serves as an annotation to the planner that the intent is to do *cmd*. Actually sets up a subtask for *cmd*, but hangs a reducing subtask off it with action *method*. The *tag* (default: **#F**) serves to pick out this reduction in the name prefix for the reducing subtask.

(REDUCTION *cmd reducees r* [*tag*])**:** The *cmd* is executed, but with the subtasks specified by *reducees* reduced to a new task with action *r*. (The *reducees* must evaluate to a list of descendants of this task.) The *tag* (default: **#F**) serves to pick out this reduction in the name prefix for the reducing subtask. When the interpreter has begun *all* of the *reducees*, it then begins the reducing task, with action *r*. When this task finishes, each of the *reducees* finishes, too (in the same way, through completion, failure, or evaporation). A task may be reduced in more than one way; the alternative ways act as if they were executed combined in a **PURSUE** — when one succeeds or fails, the others evaporate.

(SCOPE *b e p*)**:** Defined by macro to be exactly equivalent to:

```
(SEQ (WAIT-FOR b)
     (PAR (WAIT-FOR e)
          p))
```

It can succeed or fail only if b is true or becomes true. After that, it succeeds if and only if p succeeds or e is true or becomes true before p fails. In the usual case, p is "policy-like," that is, either fails or goes on forever, and the **SCOPE** construct allows p to constrain only the interval from when b becomes true to when e becomes true.

(SEQ a_1 a_2 ... a_n**)**: Do each action in order.

(:TAG l a**)**: This is not an executable form. It's equivalent to a, but if it occurs in a top-level plan or procedure body, then l (a symbol) will be bound to the task for a at the highest level in the text where there is a single task for a, i.e., the highest noniterative context, or the whole text.

(TEST p a_1 a_2**)**: Execute p, then use value as in **IF**.

(TOP-LEVEL a_1 a_2 ... a_n**)**: Do the actions in parallel. Succeed when each has failed or succeeded. When one fails, however, if the value of the Lisp variable **AGENT-REPORT*** is non-**#F**, then it is a procedure which is called with two arguments: the task for the a_i that failed, and a description of the failure as generated by the **FAIL** that initiated it. **AGENT-REPORT*** is also called when one of the a_i succeeds. In this case, the second argument is the value of a_i.

(TRY-ALL a_1 a_2 ... a_n**)**: Do actions in parallel. Succeed if one succeeds (the others evaporate); fail if all fail. The failure description in this case will be a **composite-failure** whose **COMPONENTS** is a list of all the nontrivial faildescrips for the subtasks.

(TRY-IN-ORDER a_1 a_2 ... a_n**)**: Try a_1. If it fails, try a_2, etc. Succeed when one succeeds; fail if they all fail. The faildescrip is as for **TRY-ALL**.

(TRY-ONE (c_1 a_1) ... (c_N a_N)**)**: Evaluate c's. Pick randomly from those that come up true and do the corresponding action a_i. If none true, fail. The default action can have c = **ELSE**.

A Firbyesque RAP (Firby 1989) can be defined thus (roughly);

```
(DEF-INTERP-PROC RAP (...)
   (N-TIMES 2
    UNTIL  success
      (TRY-ONE  -tests and methods-)))
```

(VALUES e_1 e_2 ... e_n**)**: Evaluates the e_i and returns the resulting values. These can be intercepted by using multiple variables in **!=** and **LET***.

(VALVE-RELEASE p v**)**: Declare that process p no longer owns valve v. If p is **#F**, the innermost process at the point of execution releases the valve. **VALVE-REQUEST/VALVE-RELEASE** pairs must match up. If the same process requests a valve twice, it must be released twice. When the process ends, however, an unconditional release of all its valves occurs.

(VALVE-REQUEST p v [f]**)**: Process p seizes valve v, or waits if another process already has it. If p is **#F**, then the innermost process at the point of execution makes the request. If f is present and non-**#F**, then it is a fluent that is set to **#F** whenever *p* loses control of *v* through pre-emption before it is officially released.

(WAIT-FOR *p* **[| :KEEP-CONTROL | :ALLOW-INTERRUPTS])**: *p* should evaluate to a fluent. Wait until *p* becomes true. If **:KEEP-CONTROL** is present, then the waiting is "active." That is, while this thread is waiting it still keeps control (as if it were implemented by virtual threads). If **:ALLOW-INTERRUPTS** is present, or there is no flag, then the **WAIT-FOR** thread is considered blocked. If all a processes' threads block, it is a candidate for pre-emption. If *p* is true initially, then no interruption occurs, even for an instant.

(WAIT-TIME *t* **[| :KEEP-CONTROL | :ALLOW-INTERRUPTS])**: Wait for *t* seconds. The **:KEEP-CONTROL/-:ALLOW-INTERRUPTS** flag behaves as for **WAIT-FOR**.

(WAIT-WITH-TIMEOUT *p t* **[| :KEEP-CONTROL | :ALLOW-INTERRUPTS])**: A combination of **WAIT-FOR** and **WAIT-TIME**. Proceeds as soon as *p* is true or *t* seconds have passed. The **:KEEP-CONTROL/:ALLOW-INTERRUPTS** flag behaves as for **WAIT-FOR**.

(WHENEVER *p* **[| :KEEP-CONTROL | :ALLOW-INTERRUPTS]** *-body-***)**: Do *body* once for every time *p* goes from false to true (including once if it's true when the **WHENEVER** starts). The **:KEEP-CONTROL/:ALLOW-INTERRUPTS** flag determines whether the thread blocks between occasions when *p* is true. **WHENEVER** never finishes, so it should be used a top-level monitor, or used inside a context that can evaporate, such as a **WITH-POLICY**.

(WITH-POLICY *p -body-***)**: Do *body*, but also do *p*, never allowing *body* to run unless *p* is blocked. Fail if either *body* or *p* fails; succeed if *body* succeeds.

(WITH-TASK-BLOCKED *t -body-***)**: Do *body*, making task *t* and all its subtasks wait until *body* is done.

(WITH-VALVE *v -body-***)**: Create a new process with a task to execute *body*. This process will request *v*, and hence own it during the execution of *body*.

4.3 Useful Lisp Functions

The following Lisp functions are especially useful in RPL plans:

(BEGIN-TASK *t***)**: A Boolean fluent that gets value **#T** when task *t* becomes active.

(CHANGED-FLUENT *f compare-fcn***)**: A fluent that is momentarily set to **#T** (as if **PULSE**d) whenever the value of fluent *f* changes. The *compare-fcn* is an equality tester that is used to tell whether the value has changed.

(CONCLUDE *f***)**: Short for **(SETF (FLUENT-VALUE** *f***) '#T)**.

(CREATE-DESIG *string property-pairs***)**: Create a new desig, which will print thus: **!:***stringK*, where *K* is an identifying numerical tag.

(CREATE-FLUENT *pat v***)**: Create a new fluent, with initial value *v*. The *pat* is for mnemonic purposes in printing the fluent.

(CREATE-VALVE *name p***)**: Create a new valve, pre-emptible if p =**#T**.

(DBG-FORMAT *s control -args-***)**: Just like Common Lisp's **FORMAT** for producing output to stream *s*. However, the output is suppressed during a run unless the global variable **INTERP-DBG*** is **#T**. It's suppressed during projection unless the variable **PROJECT-DBG*** is **#T**.

(DESIG-GET *d p***)**: Retrieve the value of property *p* from the property list of desig *d*. The search starts with the latest version of *d*, and, if no entry is found on the property list there, continues with earlier versions. If no entry is found for any version, then **#F** is returned. **(SETF (DESIG-GET** *d* ...**)** ...**)** alters the property list of the latest version of *d*.

(DRIVE *f e***)**: Connect fluent *f* to fluent *e* so that from now on the value of *f* tracks the value of *e*. Note the difference between **(DRIVE** *f e***)** and **(!=** *f e***)**. The latter transfers the current value of *e* to *f*. **DRIVE** links the two together so that future changes to *e* are transmitted to *f*. (This locution is lifted from Gat 1991.)

(FLUENT-VALUE *f***)**: The value stored in fluent *f*. Changeable using **SETF**.

(END-TASK *t***)**: A Boolean fluent that gets value **#T** when task *t* becomes finished, evaporated, or failed.

(EQUATE *new-desig old-desig***)**: Make *new-desig* be the most recent version of *old-desig*. All later requests for properties of *old-desig* or its earlier versions will start with the property list of *new-desig* and work its way back.

(MATCH-PROTECTIONS *pattern***)**: Scan list of currently active **PROTECTION** tasks for one with a *prop* matching *pattern*. Return a list of association lists, one for each match found. The entries in the list specify the values for variables in *pattern*.

(PATH-SUB (-*name-prefixes*-**)** *task***)**: The syntactic sub-sub-. . .-task of *task* arrived at by using the elements of (*name-prefixes*) in reverse order.

(PULSE *fluent***)**: Set *fluent* to **#T** and then immediately to **#F**. Any thread waiting for *fluent* to be true will be triggered.

(SET-VALUE *f v***)**: Sets value of fluent *f* to *v*. Equivalent to **(SETF (FLUENT-VALUE** *f***)** *v***)**. In RPL code, this can be abbreviated **(!=** *f v***)**, but *not* within Lisp code, where **!=** is just an abbreviation for **SETF**.

(SETF *f e***)**: Change the value of expression *f* to the value of expression *e* (not a RPL command). Currently *f* must be an expression, not a plan variable; using **SETF** on a symbol will set the global Lisp value of that symbol, not its local RPL value.

(STATE *pat***)**: Short for **(CREATE-FLUENT** *pat* **'#F)**.

(SUB -*name-prefix*- *task***)**: The syntactic subtask of *task* with the given *name-prefix*. Only the *task* argument is evaluated. (It typically is the label of a **:TAG**, which is bound to the task so tagged.)

(TAGGED *label task***)**: The sub-. . .-task of *task* **:TAG**ged with the given label.

4.4 Subtask Naming Conventions

This section contains a list of all subtask name prefixes. In the following table, if a construct says it is *"Defined by macro,"* that means its subtasks' names depend on how the macro expands. It is a good idea to use tags to avoid having to know what the macro expansions are.

If a construct has a *-body-* in the listing of Section 4.2, then there will be exactly one subtask corresponding to the sequence *body*. If it has more than one element, then a SEQ is wrapped around it internally, and an extra layer of (STEP *i*) name prefixes will be needed to get at the subtasks.

(REDUCE *cmd* ...) and (REDUCTION *cmd* ...) have just one subtask, corresponding to the *cmd* argument, with name prefix (DOIT). However, they set up task reductions for the *cmd* task or a group of its subtasks. If R is a reducing subtask of T, then it may be referred to as if it were a syntactic subtask of T with name prefix (REDUCTION [*tag*]), where the *tag* is obtained from the corresponding REDUCTION or REDUCE command. The *tag* is optional if no ambiguity is possible, i.e., if T has just one reduction.

Command	*Subtask Name Prefixes*
!=	(SETTER-VAL)
ESTABLISH-CONTEXT	*Defined by macro*
FAIL	*No subtasks*
FILTER	(FILTER-DO)
IF	(IF-ARM TRUE), (IF-ARM FALSE)
RPL-procedure body	(PROC-BODY)
LET	(STEP *i*)
LET*	(STEP *i*)
LOOP	(ITER *i*)
Loop body	(STEP *j*)
NO-OP	*No subtasks*
N-TIMES	As for LOOP
PAR	(BRANCH *i*)
PARTIAL-ORDER	(STEPNODE *i*)
PERIODICALLY	(OCC *i*)
PLAN	*Defined by macro*
PRIORITY	(REPRIORITIZED)
PROCESS	(PROCESS-BODY)
PROTECTION	*Defined by macro*
PURSUE	(PURSUIT *i*)
REDUCE	(DOIT)
REDUCTION	(DOIT)
Reduced task	(REDUCTION [*tag*])
SCOPE	*Defined by macro*
SEQ	(STEP *i*)
TEST	*Defined as a macro*
TOP-LEVEL	(COMMAND *i*)
TRY-ALL	(ALT *i*)
TRY-IN-ORDER	(ATTEMPT *i*)
TRY-ONE	(CLAUSE *i*)
VALUES	*No subtasks*
VALVE-RELEASE	*No subtasks*
VALVE-REQUEST	*No subtasks*
WAIT-FOR	*No subtasks*