# COLBERT: A Language for Reactive Control in Saphira

Kurt Konolige
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
konolige@ai.sri.com
http://www.ai.sri.com/~konolige/Saphira

1. Controlling a Robot

What does it mean to write programs to control a robot? Robots can sense the world and act within it; so, in general, a robot control program is one that takes the robot's sensory input, processes it, and decides what motor actions the robot will perform. But the mapping between inputs and outputs is a very complex one, and the control task requires some decomposition into simpler elements to make it workable. In recent years there has been some convergence on an architecture for autonomous mobile robots. In general form it looks something like the diagram in Figure 1-1.
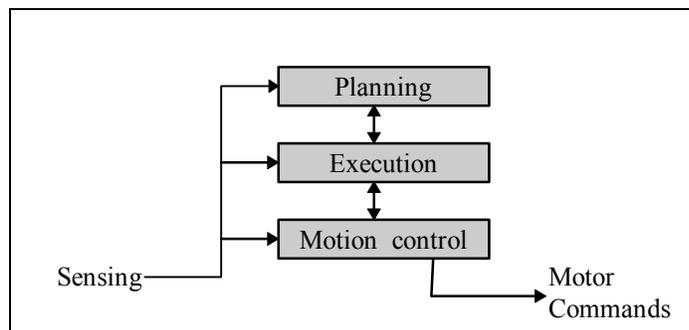
Figure 1-1 A hybrid control architecture

The bottom layer is a controller that implements some form of motion control for the robot. This layer can be quite complex; for example, in the Saphira architecture it consists of a fuzzy controller that implements a set of behaviors for achieving goals such as corridor following, obstacle avoidance, and the like [Kon97]. The second layer is a

sequencer that initiates and monitors behaviors, taking care of temporal aspects of coordinating behaviors, such as deciding when they have completed their job, or are no longer contributing to an overall goal, or when environmental conditions have changed enough to warrant different behaviors. The sequencer must complete its computations in a timely manner, although not as quickly as the control layer. In the top layer, long-term deliberative planning takes place, with the results being passed down to the sequencing layer for execution. Generally, the planner is invoked and guided by conditions in the sequencing layer, e.g. a task failing or completing.

There are many different instantiations of this architecture, including Saphira [Kon97], SSS [Con92], ATLANTIS [Gat92], RAPs [Fir94], AuRA [Ark90], and Payton's reactive planners [Pay90]. In almost all of these, the sequencer plays the role of the main executive, taking advice from the planner and invoking behaviors to accomplish goals. When one thinks of writing robot programs, it is sequencer programs that are the result. In fact, it's possible to think of the planner as an automatic generator of robot programs, which are then executed by the sequencer.

Most of architectures mentioned in the previous paragraph concentrate on the interaction between the layers, how to integrate behaviors, sequencers, and planners. In contrast, in this paper we are concerned with how a user can write sequencer programs to effectively control the robot. Our emphasis is on issues of language and semantics: what is a good language for robot programs, what kind of semantics is appropriate for the sequencer, and how does the language fit the semantics. The result of our inquiries is the sequencer language Colbert, a part of the Saphira architecture.

Colbert draws on two sources for its concepts. The first is finite state automata (FSAs) [Hop79]. FSAs are ubiquitous in computers and robotics, because they provide a way of defining a mapping between the internal state of a automaton and its operation in the world. When you drop coins into a soda machine, its internal state changes, until it gets to a state in which you've paid enough; then it drops a soda. FSAs are a great way to encode procedural knowledge: knowledge of how to achieve some goal. This is especially true when the procedure includes *conditional actions*, which must test the state of the environment to make a decision about which action to perform next. In Colbert, a program is an *activity* whose semantics is based on FSAs.

The second source of inspiration is from concurrent processes. Complex robot control problems are often best decomposed into sets of concurrent processes that communicate and coordinate their activity. In Colbert, a set of activities executes concurrently to achieve a goal. Activities have a hierarchical structure (one activity can spawn another, and is its *parent*). Activities communicate through a global data store, and by sending each other signals.

Having an adequate semantics doesn't mean that control programs will be easy for a user to write or debug. In fact, writing FSA structures directly is not a pleasant task, and all modern computer languages use implicit sequencing and explicit looping statements to define the flow of execution control. It would be nice to use a language that has familiar control structures, so that users would not have to learn another programming language. With this in mind, we chose for Colbert a subset of the ANSI C language, along with a few extensions for robot control. Surprisingly, even though the semantics

is based on FSAs, there is relatively little that a C programmer must learn to begin writing correct robot control programs.

A second user concern is the ability to debug and edit control programs as part of the development cycle. A problem with developing using C is that the compile-load-debug-edit-recompile cycle is tedious, and getting back to where the mistake occurred can be time-consuming or even impossible when dealing with robots operating in a real-world environment. All of these issues indicate that an interpretive development environment is desirable, where errors are signaled, the user can examine the state of the system, make changes to programs, and continue with the changed program. We have implemented a Colbert evaluator that executes source language statements directly, so that programs can be modified during execution. The evaluator also allows the user to probe the state of the system during execution to determine where errors occur, and to load compiled C code for efficient execution of compute-intensive routines as part of an activity.

Finally, we have tried to make Colbert efficient and portable to most operating systems. The evaluator is fast enough to be used for production robot programs; but it is also possible to compile Colbert activities into native C code for even more efficient execution. Since the Colbert executive is written in C, and requires only minimal support from the OS, it can run under most OS's: we have implemented versions on most Unix systems, and on Windows 95/NT.

2. Activity Examples

Activities control the overall behavior of the robot in several ways.

- Sequencing the basic actions that the robot performs.
- Monitoring the execution of basic actions and other activities.
- Executing activity subroutines.
- Checking and setting the values of internal variables.

In this section we'll look at some simple examples of robot behavior. These examples will illustrate the functionality of activities in controlling robot behavior. A more detailed discussion of the syntax and semantics is in subsequent sections.

2.1 Patrol

We want the robot to patrol up and down between two goal points, repeating this activity a specified number of times. The basic actions the robot can perform are (1) turning to a heading, and (2) moving forward a given distance. For this example we won't worry about the problem of *robot localization*, that is, how the robot will maintain registration between its internal map (the two goal points) and the external world.

The simplest way to realize the patrol activity is as a perpetual `while` loop, in which the primitive turn and forward motion actions are executed in sequence. Here is the proposed activity schema:

```
act patrol(int a)
{
    while (a != 0)
    {
        a = a-1;
        turnto(180);
        move(1000);
        turnto(0);
        move(1000);
    }
}
```

Figure 2-1 A simple patrol activity

This simple example illustrates three of the basic capabilities of the Colbert control language. First, the two basic actions of turning and moving forward are sequenced within the body of the `while` loop. As each action is initiated, an internal monitor takes over, halting the further execution of the `patrol` activity until the action is completed. So, under the guidance of this activity, the robot turns to face the 180$^\text{o}$ direction, then moves forward 1000 mm, then turns to the 0$^\text{o}$ direction, then moves forward another 1000 mm. The net effect is to move the robot back and forth between two points 1 meter apart.

The enclosing `while` loop controls how many times the patrol motion is done. The local variable `a` is a parameter to the activity; when the activity is invoked, for example with the call `start patrol(4),` this value is filled in with an integer. On every iteration, the `while` condition checks whether `a` has been set to zero; if not, the variable is decremented and the loop continues. (Note that, to make this an almost infinite loop, just invoke `patrol` with a negative argument.) Using the variable `a` to keep track of the number of times the movement is done illustrates the capability of checking and setting internal variables, which can be very handy even for simple activities.

The language of activities is based on ANSI C. When an activity schema is defined, the keyword `act` signals the start of the schema. The schema itself looks like a prototyped function definition in C. Constructs such as local variables, iteration, and conditionals are all available. In addition, there are forms that relate specifically to robot action. In this case, the actions are primitive motions available to the robot: turning and moving forward. When the activity schema is invoked, an *activity executive* interprets the statements in the schema according to a finite state semantics. Basic actions cause the executive to wait at a finite state node until the action is completed (or some escape condition holds, such as a timeout). So, while the activity schema looks like a standard C function, its underlying semantics is based on finite state automata for robot control. The user, who typically wants to sequence robot actions in the same way as he or she would sequence computer operations, can write control programs in a familiar operator language; the executive takes care of matching the activity schema statements to the finite state automaton semantics, so that the intended robot behavior is the result.

2.2 Surveillance Robot

While sequencing basic actions is the typical evaluation mode, the language also supports concurrent execution, in which several activities working in parallel coordinate the robot's actions. Suppose we want to program the robot to patrol until it sees some object in front; then it should stop patrolling and approach the object. To accomplish this task, we'll set up two activities: the patrol activity of the previous example, and a supervisory activity that checks if there is something in front of the robot, and if so, approaches it.

```
act approach()
{
   int x;
   start patrol(-1) timeout 300 noblock;
 checking:
  if (timedout(patrol) || sfStalledMotor(sfLEFT))
    fail;
  x = ObjInFront();
  if (x > 2000) goto checking;
  suspend patrol;
  move(x - 200);
  succeed;
}
```

Figure 2-2 An activity that monitors another

This activity starts off by invoking `patrol` with a negative argument, so it continues indefinitely. However, instead of causing the `approach` to wait for its completion, the `patrol` activity is invoked with two special parameters. The first, `timeout 300`, causes `patrol` to quit after 30 seconds (300 cycles) have elapsed. The second, `noblock`, allows the execution of `approach` to continue in parallel with `patrol`. The former now goes into a monitoring loop, in which it checks for objects in front, for a motor stall, and for the state of the `patrol` activity. If it determines that `patrol` has timed out, or if a motor stalls (indicating the robot ran into something immovable), then `approach` exits in a failure state. The activity executive keeps track of the dependencies among activities; in this case, since `approach` called `patrol`, exiting `approach` automatically exits `patrol`. Thus, if the motor stalls, all activity started by `approach` will be suspended.

If, on the other hand, `approach` determines that there is an object less than 2 meters in front (by calling the perceptual routine `ObjInFront`, which returns the distance to the nearest object), then it suspends the `patrol` activity, and moves to within 20 cm of the object. The patrol activity must be suspended, otherwise the `Move` action will conflict with the actions being issued by `patrol`. After the robot moves near the object, the `approach` activity exits with the success state.

In this example, two activities execute concurrently, and coordination is achieved by *signals* that are sent between them. Activities can examine each others' state, and take appropriate action. As the monitoring activity, `approach` has the responsibility of

checking the state of `patrol` to see if it has timed out, and also checking for other conditions that would cause the suspension of `patrol` and the initiation of new behavior. Finally, if `approach` is itself part of a larger activity, then by exiting with success or failure, it can signal other activities of its result.

The use of a C-like language, together with a concurrent finite state semantics, makes it easy to write complex control routines in a few simple lines. In the next sections we'll examine the semantics of Colbert in more detail, and show how the executive interprets the Colbert language.

3. Language and Semantics

Finite state automata (FSAs) are an ubiquitous paradigm in computer science and electronics engineering. FSAs are used to program logic chips, to design microprocessors, to run soda machines, and to reason about the computational complexity and decidability [Hor89]. They have also been the inspiration for several robot control languages, including the *situated automata* of Kaelbling and Rosenschein [Kae90], and the circuit semantics of ATLANTIS [Gat92]. We combine the basic structure of FSAs with ideas from concurrent systems to produce a semantics for Colbert.

3.1 Finite State Automata

A finite state automaton consists of:

- sets S (states), I (inputs), and O (outputs)

- a transition function $f(s,i) \rightarrow s$ from states and inputs to states

- an output function $g(s) \rightarrow o$ from states to outputs

It's convenient to represent FSAs using arcs and nodes: nodes are the states, and arcs are the transitions between states. The arcs are labeled with the transition condition necessary for taking the arc to the next state, and states are tagged with their output function. Figure 3-1 shows the FSA for the `patrol` activity defined in the previous section. The transition function label is in boldface, the output label in italics.
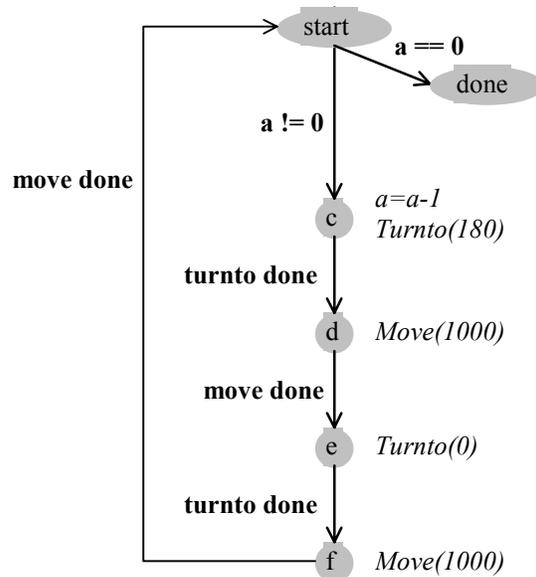
Figure 3-1 The finite state automaton for `patrol`

The first thing to note about the FSA is that its states don't correspond exactly to the statements in the activity. For example, the `while` statement has been translated into a set of nodes (*start, done, c*) which split the condition of the loop. In general, conditional and looping statements in Colbert will translate to a set of nodes with conditional labels in the FSA.

Actions at the nodes include primitive robot actions and internal state changes. In pure FSAs, all state information is encoded in the states themselves. For Colbert, the nodes represent only the state of the activity; other robot state information is handled separately (and more efficiently) as part of the Saphira perceptual space.

In the activity schema, no wait conditions for primitive actions were given explicitly. In the FSA, these conditions are given as the conditions for transition to the next state. When an action command is issued, the FSA waits in the issuing state until the action is finished. This default translation can be changed by the addition of the `noblock` and `timeout` parameters in Colbert.

Note that the output function associated with a node is performed only once, when control first arrives at the node. All self-transitions back to the node (which are not explicitly drawn in the figure) do not result in the output function being called again.

The strength of the Colbert language lies in the ability to make an intuitive translation from operator constructs in C to FSAs capable of controlling the robot. FSAs can be tedious to program directly, because straightforward sequences and loops that are typical of most programs translate into lengthy sets of nodes and arcs with a

linear or looping structure. Consider trying to write in C, where after each statement you have to say which statement to go to next! In addition, common FSA constructs, like waiting for actions to finish, can be assumed implicitly as part of the semantics of Colbert, rather than written explicitly in the construction of the FSA.

3.2 Colbert Statements

Colbert statements are from one of four categories:

1. Control actions
2. Activity state tests
3. Internal state actions
4. Sequencing actions

Table 3-1 lists the statements available in these categories. We've already seen examples from each of the categories in the `patrol` and `approach` activities. The sequencing and internal state actions comprise the standard C portion of the language. C assignments and function calls have their normal interpretation, changing the state of internal C variables. Sequencing actions, which include typical C iteration operators, are translated into a set of FSA states with appropriate branches, as in the while statement of the patrol activity.

Control Actions	Example	Description
Primitive Action	`move(1000) timeout 30;`	Start a primitive action
start act	`start patrol noblock;`	Start an activity
<signal> act	`suspend patrol;`	Signal an activity
Activity State Tests		
<state>(act)	`sfGetTaskState("patrol")`	Test the state of an activity
Internal State		
C assignments and functions	`x = ObjInFront()+10;`	Test or set the state of the database
Sequencing Actions		
goto	`goto start;`	Go to a state
while, if	`if (a == 0)` `  goto start;`	Iterative and conditional execution
waitfor	`waitfor(timedout(act)` `        ||a<0);`	Conditional suspension
wait <int>	`wait 30;`	Wait *n* cycles

Table 3-1 Colbert statement summary

Control actions translate to a single FSA node for executing the action, and a transition based on the completion of the activity or action. If noblock is asserted, then the transition is taken immediately; if a timeout is asserted, then there is an additional transition based on the timeout value.

Control actions can also change the state of other activities, by sending them signals. Similarly, an activity can accept signals from other activities, changing the state of the underlying FSA.

Activity state tests aren't statements *per se*, but are expressions that can be used where C expressions are normally allowed. They allow conditionals to check for the state of another activity.

Finally, intentions can modify the state of their execution using various sequencing operators: goto, iteration, conditional, and suspension operators.

3.3 Subactivities

Colbert supports an execution model in which activities may be invoked as children of an executing activity. This capability supports hierarchical task decomposition, an important method for robot control [Fir94, Ark90]. Consider the task of moving an object from one place to another (taken from [Fir96]). It's natural to decompose this into three subtasks: picking up the object, going to the destination, and dropping the object. In Colbert, we would write the following activity.

```
act move object(int dest)
{
    start pickup;
    start goto(dest);
    start drop;
}
```

Figure 3-2 An activity with subactivities

The subactivities pickup, goto, and drop are executed in turn. The move_object activity stops at each until it finishes, then goes on to the next. This default execution model is the same as for primitive actions. However, there are cases in which hierarchical decomposition is not the best way to accomplish a task; rather, it is useful to have several activities executing concurrently. This behavior is typical with monitoring tasks, which can stretch over several subactivities. Firby [Fir96] gives the example of monitoring the robot's gripper during the move_object activity. If the robot drops the object, it should attempt to pick it up before continuing to the destination. In Colbert, a monitoring activity can be started and run concurrently with the other subactivities, as in the surveillance example of Figure 2-2. In the current example, we start a monitoring activity using the noblock option.

```
act move object(int dest)
{
    start monitor hand noblock;
    start pickup;
    start goto(dest);
    start drop;
}
```

Figure 3-3 Monitoring the `move_object` activity

Figure 3-4 shows the execution structure of this activity. The execution of the schema starts at *a*, and invokes the monitoring activity (which starts at node *n*). At *b* the `pickup` subactivity is invoked, and the transition to node *c* doesn't take place until the subactivity finishes. During the execution of pickup, there are three concurrently executing activities: `move_object`, `pickup`, and `monitor_hand`. When the toplevel activity `move_object` is exited, the Colbert executive automatically reaps any subprocesses, so the `monitor_hand` process is terminated.
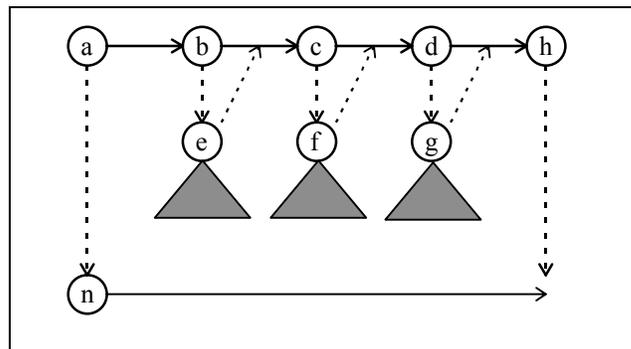
Figure 3-4 Activity execution structures

This example is incomplete, because we haven't shown how the monitoring process interacts with its siblings to achieve the task in the presence of carrying failures. In the next section we look at the coordination mechanisms available to concurrent tasks.

The hierarchical structure of activities is very much like the child process structure of Unix systems. What distinguishes Colbert activities is the FSA nature of their semantics, which makes the coordination process more easily understandable and controllable.

3.4 Concurrent Activities and Synchronization

Often the task of robot control can be decomposed into a set of subtasks that are mostly independent, but require some form of coordination. Colbert's semantics

supports a set of concurrent activities that communicate indirectly through a global database, and directly by sending signals. The FSA nature of activities is handy for communicating state information.

Although an activity can be queried to see if it is in any given state, there are some predefined states typically used for signaling. These are listed below in Table 3-2.

State	Meaning
sfINIT	Initial state of an activity
sfSUCCESS sfFAILURE	Termination states: activity succeeded or failed in its goal
SfTIMEOUT	Termination state: activity timed out
sfSUSPEND	Suspended state: the activity is not running
sfINTERRUPT	State for activities after an interrupt signal
sfRESUME	State for activities after a resume signal

Table 3-2 Standard signals for activities

When an activity is first started, it is set to the sfINIT state. Typically this is the first statement of the activity, but an activity can specify a particular start position by using the onInit label. For example, the following activity starts in the middle:

```
act aa(int x)
{
  loop:
   if (x == 0) succeed;
  oninit:
   x = x-1;
   goto loop;
}
```

If an activity falls through to the end, it is considered to have succeeded. Otherwise, the activity can terminate itself and signal success or failure by using the special actions succeed and fail. The activity can also suspend itself by using the suspend action. In the suspended state, no further processing takes place until another activity sends a signal, usually the resume signal.

Interruption and resumption are the normal way in which activities are requested to stop and start their processing, once invoked. An interrupt signal causes an activity to go to the special oninterrupt label. There, the activity should clean up anything that needs it, such as terminating current movement actions, and then suspend itself. On resumption, the activity should re-establish any state it needs, then continue its processing. Here is an example of making the patrol activity interruptable and resumable. When patrol2 is interrupted, it first waits for any forward motion to be finished (sfDonePosition() returns 1 when any Move command is finished). Then it suspends itself. This means that the robot finishes up at one of the patrol

endpoints. On resumption, the counter is incremented, and the patrol continues. Note that this is not a perfect solution, since the robot could have stopped at either point, and may resume by patrolling ahead of or behind its original path. But it does illustrates the idea of interruptability.

```
act patrol2(int a)
{
  start:
   while (a != 0)
   {
      a = a-1;
      turnto(180);
      move(1000);
      turnto(0);
      move(1000);
   }
   succeed;
  oninterrupt:
   waitfor (sfDonePosition());
   suspend;
  onresume:
   a = a+1;
   goto start;
}
```

Figure 3-5 An activity that responds to interrupts

Activities can also be coordinated with global variables, which are visible to all activities. For example, the approach activity waits for a condition in which there is an object close by. The function ObjInFront() could invoke a perceptual routine, or could just check the value of a variable that another activity was responsible for setting.

One problem that all concurrent systems have is coordinating their accesses to global variables. Several processes may attempt to change the same variable at the same time: for example, a process may be executing the statement x=x+1, while another is changing x. The result may not be to increment x. The same coordination problem exists with signals: one process may attempt to interrupt another when it is executing a statement, and the statement may not be fully executed, leaving the process in an indeterminate state.

In typical concurrent systems, there are coordination mechanisms for dealing with these problems: critical sections, mutexes, and the like. In Colbert, the FSA semantics provides a natural coordination mechanism. Activity transitions are executed *synchronously*, and signaling takes place when all activities are settled at an FSA node. Synchronous behavior is enforced by the execution model, explained in the next section.

Let's return to the move_object example (Figure 3-3) and add coordination signals to the activities. A partial listing is shown in Figure 3-6. As before, the monitor_hand activity is started in nonblocking mode, and immediately sent the suspend signal so that it isn't active until pickup actually grasps the object, at which

point it is sent the `resume` signal. Now monitoring is active throughout the robot's activities, until during `drop` it gets sent a `suspend` signal.

```
act move object(int dest)
{
   start monitor hand noblock suspend;
  restart:
   start pickup;
   start goto(dest);
   start drop;
   succeed;
  oninterrupt:
   suspend monitor hand;
   goto restart;
}

act pickup
{
   gripper down;
   gripper grasp;
   resume monitor_hand;
   gripper up;
}

act drop
{
   suspend monitor hand;
   gripper ungrasp;
   gripper down;
}
```

Figure 3-6 Signaling in the `move_object` activity

Note that there is no special processing needed for simple failure recovery. If `monitor_hand` detects a gripper failure, it just sends an interrupt signal to `move_object`. Since all children of `move_object` will also see this signal, whatever subactivity is occurring will also be interrupted. On interrupt, `move_object` resumes by again trying to grasp the object and continue to its drop-off point.

In this example, it was possible to refer to executing activities using their schema names. In more complicated cases, the same schematic activity may be invoked several times, giving rise to an ambiguous reference in signaling. Colbert allows an activity to be given an *instance name* when it is invoked. In this way, multiple invocations of the same schema can be kept separate for signaling purposes. This is not a very sophisticated scheme, however. A more useful but still simple method would be to allow activity instances to be members of user-defined classes, e.g., a *monitor* class, or even a *monitor_gripper* class. The user could create classes and subclasses as needed, and signal an activity based on its name, its name and class, or just its class.

4. Colbert Executive

In this section we describe how the Colbert executive implements the FSA semantics of activities. To start an activity schema, it must be invoked with the `start` command, which puts an instance of the schema onto Colbert's structured list of activities. The Colbert executive has the job of cycling through the activities and executing them incrementally, giving an operational meaning to the underlying FSA semantics.

4.1 Synchronous FSA Cycle

One of the problems faced in implementing the executive is that it must work the same on a number of different OS platforms (MS Windows and different flavors of Unix), all of which have different support for concurrent activity. Another problem is that a Saphira client can have tens or even hundreds of activities executing concurrently, so the overhead of servicing them must be low. This rules out expensive OS implementations such as separate processes or even threads. Instead, we take advantage of the discrete nature of the FSA to update activities in a round-robin fashion. The executive uses a native timer mechanism to schedule an interrupt every 100 milliseconds, which is its basic cycle time. The cycle time is short enough to ensure timely response to new conditions, while being long enough not to load current processors excessively.

Within this cycle, every executing activity will progress through at least one state of its underlying FSA. The executive cycles through the activity list, and for each activity, it evaluates statements until it reaches a halting condition based on the FSA semantics.

The halting conditions depends on the state of the activity. One of several things may happen.

1. If the current state is waiting for a condition to occur, and that condition is not satisfied, the activity stays in the state and returns. Typically, waiting conditions are issued explicitly with the `waitfor` command, or implicitly by primitive actions or activity calls.

2. If the current state is not waiting, then the current statement is evaluated. Depending on the statement, the executive may halt evaluation and move on to the next activity, or it may continue to evaluate successive statements.

Statements which cause an execution break are:
- `goto`
- the last statement in a `while` body
- the condition of a `while` statement being false
- `waitfor`
- `start`
- any signaling action
- any primitive action

These halting conditions are meant to make ordinary C operations efficient, since they can execute sequentially without causing a break. For example, the following sequence of statements will not cause a break until either the `succeed` or `goto` statement is executed.

```
   if (x == 0) succeed;
 oninit:
  x = x-1;
  goto loop;
```

By forcing multiple statements to be evaluated in a single execution cycle, the executive preserves the efficiency of sequential execution, while still allowing an activity to break at critical junctures. When the executive has finished with an activity's evaluation cycle, the activity is at an FSA node, with all output functions having been executed.

Because each activity is executed in turn, and execution always finishes at an FSA node, all activities are executed synchronously (and sequentially). Thus, there is no problem of race conditions or competing update among concurrent processes. However, because activities are executed sequentially rather than concurrently, there can be order-dependent phenomena that are unexpected. For example, to propagate a signal sequentially through 4 activities could take 1, 2, 3 or 4 cycles, depending on the execution order of the activities within the cycle.

4.2 Executive Structure

Figure 4-1 shows the structure of the Colbert executive. The main data structure, the activity list, is a structured collection of current activities. One way to think of these activities is as a set of threads in an operating system. Each of the threads is a separate execution module, and all threads share global variables. In addition, there is a hierarchical structure among the threads.

The basic cycle is for the executive to look at each activity in the activity list, check to see if its state can change, invoke the requested actions, and update the state. All of this happens within the 100ms basic cycle time, so the response to new conditions is relatively quick.

For each activity, the executive checks if it is in a waiting or suspended state; if so, it bypasses execution of this activity. If not, it evaluates the activity until its next halting condition, as described in the previous section. In addition, it checks for a timeout condition, and suspends an activity or cancels an action if it exceeds its limit.

An executed action can result in a signal being sent, or a new activity or action being invoked. The executive handles these by issuing the appropriate commands. For activity invocation, the executive looks up the activity schema in its library, instantiates any arguments, and adds the activity to the activity list. In the blocking case, the executive is responsible for checking the blocking and timeout conditions of the subactivity, and resuming the calling activity when appropriate. In the non-blocking case, the executive starts the subactivity as a concurrent activity, and maintains the link to its parent. If the parent is signaled (for example, with an interrupt or resume signal), then the appropriate

signal is also passed down to the child. The executive also handles requests from other Saphira processes for activity invocation or signaling.
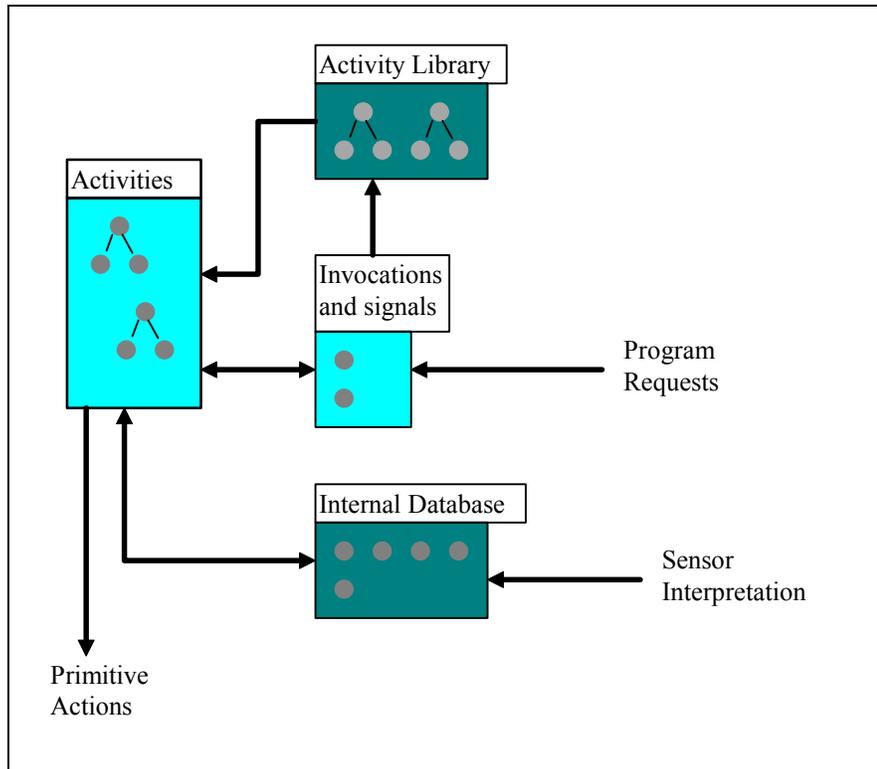
Figure 4-1 Major components of the Colbert executive

In Colbert, there is only one way in which an activity can be invoked, by calling it directly with its arguments. In PRS-Lite, more advanced automatic invocation of activities is possible, through the use of a database of goals and facts [Geo89]. In this mode, environmental conditions or the posting of a goal can trigger the invocation of an activity. It would not be difficult to add this capability to Colbert, and we intend to do so in the future.

When an activity finishes, either explicitly by signaling its success or fail state, or implicitly by falling through its last FSA node, all its children are sent `suspend` signals by the executive. Thus, an activity that finishes has no executing subactivities: the executive enforces the hierarchical nature of subactivities.

4.3 Implementation

The Colbert executive is implemented in C and requires only the services of a 100 ms interrupt. It uses global and local data structures to store activity closures (activities

with their variable bindings). Compiled activities use C's native variable storage, so variable access is very fast. For either compiled or evaluated activities, the context switch between activities is also very fast. On typical processors (100 MHz Pentium) we have measured times on the order of 10 μs. A typical Saphira client will have some 40 compiled activities and 10 interpreted activities running concurrently, all executing within the 100 ms basic cycle. In the normal case these activities use only a small fraction of available CPU resources, on the order of 5% for a 100 MHz Pentium, and less for more powerful processors such as those in current Sparcstations or SGI machines.

The complete Colbert executive has a very small footprint. Although it was not originally intended to run on embedded systems with limited memory, there is no reason it could not do so.

The compilation of activities takes place in two phases: first, an activity is translated into a C function that implements its underlying FSA, and then the C function is compiled. The translation is relatively straightforward: turning iterations into looping FSA structures, conditionals into branching structures, and so forth. One interesting aspect of the translation is that a static analysis can determine the execution halting points, which can then be incorporated directly into the C procedure. When the executive invokes the procedure, it returns exactly at such a halting point.

Evaluation of activities is more difficult. An activity schema, in text form, is parsed using a YACC front-end and converted into an internal form suitable for evaluation. The parser must recognize a subset of ANSI C expressions and statements; this subset has been simplified to remove complex typing and few other constructs. At this stage all conversion of textual variables to internal pointers is done, as well as linking to internal C functions and variables of Saphira. There are also some translations to turn iteration constructs into a form that more closely resembles their FSA semantics, as in compilation. The internal form of activities is interpreted by the Colbert evaluator when an activity is invoked.

Because the evaluator is written in YACC, the evaluator itself is a portable C program. We have implemented it on all the systems that Saphira runs on. The parser program can be large, and may not be suitable for an embedded processor, but we have not yet made any experiments to determine if this is so.

The Colbert evaluator is available to the user at runtime, for examining the state of the system, and for invoking activities and sending them signals from the command line. All Saphira internal variables and functions are available to the evaluator, as well as any user-defined compiled C functions that are dynamically loaded into the system at runtime. The Colbert executive catches any system errors and suspends the responsible activity, so the user can examine its state and determine the problem. The evaluator makes an effective debugging tool that is portable across all implementations of the Colbert executive. It is also interesting to be able to define and evaluate C functions interactively, something dear to the heart of every LISP programmer.

5. Other Control Languages

There are a large number of languages and systems for robot control, and in this section we look at a few that are directly relevant to Colbert, especially with regard to the issue of language design for user programming.

5.1 PRS-Lite

The immediate predecessor of Colbert is the PRS-Lite executive [Mye96], a reactive controller based loosely on the Procedural Reasoning System [Geo89]. PRS-Lite shares many of the same ideas as Colbert, including a finite state semantics, and concurrent activities.   But Colbert differs in some important respects.   First, it extends the coordination component of PRS-Lite to include a hierarchical organization among the activities, and a signaling system for interruption, suspension, and resumption. Second, where PRS-Lite takes FSAs as the *language* of activities, Colbert uses C as its language, with FSAs as the underlying *semantics*.   While this might not seem like a large difference, conceptually it makes robot control programs much easier to understand and write, especially for programmers used to sequential, conditional, and iterative constructs. Finally, Colbert offers an activity (and C) evaluator, which was not available under PRS-Lite. Interestingly, the full version of PRS, written in LISP, has a graphical and textual language for activities that support the same kind of interactivity as Colbert [Wil95]. But again the language is based on FSAs, and it might be useful to import the Colbert language into PRS as a compact way of specifying activities in that system.

5.2 RAP

RAP [Fir94, Fir96] is a reactive plan execution system that shares many features with PRS and PRS-Lite. A primary emphasis of the RAP system is on modularity and reusability of activity schemas.   A RAP method (which corresponds to an activity schema) is a parameterized FSA schema for accomplishing a goal using a sequence of behaviors and other RAPs. The RAP executive manages a hierarchical tree of RAPs, decomposing more difficult tasks into sequences of subtasks. The RAP system also has a facility for concurrent execution of RAPs, along with a signaling system.

RAPs are defined using LISP data structures.   There is a limited form of conditionalization, but no iteration operators. Subtasks can be specified as operating in sequence or in parallel; the parallel operator is especially nice for spawning several subtasks that must all complete before continuing. Because it is defined in LISP, the RAP system has LISP's interactive facilities available for debugging and program development.

5.3 L

L is a commercial language for robot control based on LISP [Bro95]. It is a remarkable language, in that it can run a LISP system in an embedded system with 10 KB of memory, complete with garbage collector. To do so, it makes a number of simplifications of the LISP language. But **L** also implements a new multithreading

facility, to support concurrent execution of multiple threads of execution. And on top of this it defines a set of macros, called MARS, that facilitate interthread coordination.

Because it is a LISP, **L** lets users examine the executing system for debugging. In a typical configuration, however, **L**'s executive, VENUS, does not include either an evaluator or a compiler, so new or updated programs must be compiled externally and downloaded to the running system. This limitation stems from the use of **L** in embedded systems with very limited memory; it is conceivable that Colbert could also fit into such systems, but would have to forego the full evaluator with its large parsing program.

Colbert and **L**, despite their language differences, actually take a similar approach to the robot control problem. Like Colbert, **L**'s multiprocess scheme relies on each process being interruptable at a fine-grained level; in the case of **L**, it's at every procedure call. And they both define a signaling system for interprocess communication. But Colbert differs in using FSAs as its underlying semantics, in having a hierarchical structure for activities, and in providing support for typical invocations of robot actions. It also appears to be more efficient than **L** in its multiprocess implementation, since switching threads in **L** can be expensive.

5.4 MissionLab

MissionLab is a toolset that implements the Societal Agent theory [Mac97]. According to this theory, robot control (including multirobot control) is accomplished by recursive assemblages of behaviors. Temporal sequencing of behaviors is provided by a FSA semantics and language. Like PRS' Act editor, MissionLab provides a graphical interface with which the user can construct and debug FSAs. One of the interesting aspects of MissionLab is that assemblages are defined independent of any particular robot architecture. When an architecture is specified (e.g., SAUSAGES or AuRA [Ark90]), MissionLab generates concurrent procedures for implementing the FSAs using the action methods of the architecture.

MissionLab's strengths are the graphical user interface, the ability to bind to different robot control architectures, and the ability to specify an interaction mode for behaviors: cooperative, competitive, sequential. The use of recursive assemblages of behaviors is similar to the hierarchical structure of activities in Colbert.

5.5 GOLOG

GOLOG is a language for robot control based on the situation calculus [Lev96]. It is unique in robot control languages in having a logic-based semantics. In fact, GOLOG programs look a lot like Prolog programs with added procedural operators, and are interpreted in the same way: by a theorem prover. For example, here are two GOLOG procedures for an elevator control program:

```
proc control
    [while (∃n)on(n) do serve_a_floor endWhile];
    park
endProc

proc park
    if current_floor=0 then open
    else down(0); open
    endIf
endProc
```

Evaluating these procedures, in the presence of initial conditions for the elevator (and some other axioms about actions such as *down*) produces a sequence of primitive actions that can be executed by the elevator controller. Given the truth of the axioms, the sequence is guaranteed to fulfill the conditions of the procedures.

While the GOLOG language uses iterative and conditional constructs similar to those of Colbert, it is much more expressive. For example, there is quantification ($\exists n$) as well as nondeterministic choice. GOLOG programs can be very compact specifications of complex controllers. While theorem proving can be expensive, it takes place offline, generating a executable sequence to be used by a reactive controller.

GOLOG is an interesting experiment in high-level description for robot control. However, it remains to be seen if the rather complicated situation calculus semantics will be suitable for real world controllers. Recently, GOLOG has been extended to deal with prioritized, concurrent programs, as well as exogenous events (those not under the control of the robot) [Lev97].

6. Conclusion

The design criteria for Colbert are:

1. To have a simple language with standard iterative, sequential and conditional constructs.

2. To have a clear and understandable semantics based on FSAs.

3. To have a debugging environment in which the user can check the state of the system and redefine Colbert activities.

4. To have an small, fast, and portable executive.

The current implementation of Colbert fulfills these objectives. Whether Colbert will be successful as a robot control language remains to be seen. Currently it is only available as part of a larger robot architecture, Saphira, and so is limited to that user community. But it should be possible to adjoin Colbert to other architectures, where it would function as the sequential controller for the system. Given that Colbert programs are compact and easily transferred, we hope to build up a library of useful routines that can be shared in the user community.

References

[Ark90] R. C. Arkin, Integrating behavioral, perceptual and world knowledge in reactive navigation*, Robotics and Autonomous Systems*, **6**:105--122, 1990.

[Bro95] R. A. Brooks and C. Rosenberg, **L** - A Common Lisp for embedded systems, Association of Lisp Users Meeting and Workshop LUV (1995).

[Con90] J. Connell, SSS: A hybrid architecture applied to robot navigation, in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2719-2724, 1992.

[Fir94] R. J. Firby, Task networks for controlling continuous processes, in *Second International Conference on AI Planning Systems*, pp. 49-54, 1994.

[Fir96] R. J. Firby, Modularity issues in reactive planning, in *Third International Conference on AI Planning Systems*, Edinburgh, Scotland, pp. 78-85, 1996.

[Gat92] E. Gat, Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots, in *Proceedings of the AAAI Conference*, 1992.

[Geo89] M. P. Georgeff and A. L. Lansky, Reactive reasoning and planning, in *Proceedings AAAI Conference*, pp. 677-682, 1987.

[Hop79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Hor89] P. Horowitz and W. Hill, *The Art of Electronics*, Cambridge University Press, 1989.

[Kae90] L. Kaelbling and S. Rosenschein, Action and planning in embedded agents, *Robotics and Autonomous Systems*, **6**:35-48, 1990.

[Kon97] K. Konolige, K. Myers, A. Saffiotti and E. Ruspini, The Saphira architecture: a design for autonomy, *Journal of Experimental and Theoretical Artificial Intelligence*, **9** (1997) pp. 215--235.

[Lev96] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R, Scherl, GOLOG: A logic programming language for dynamic domains, *Journal of Logic Programming*, 1996.

[Lev97] H. Levesque, Concurrency in the situation calculus, in preparation.

[Mye96] K. L. Myers, A procedural knowledge approach to task-level control, in *Proceedings of the Third International Conference on AI Planning Systems*, AAAI Press, 1996.

[Pay90] D. W. Payton, J. K. Rosenblatt, and D.M. Keirsey, Plan guided reaction, *IEEE Trans. on Systems, Man, and Cybernetics* **20** (6), 1990.

[Wil95] D. E. Wilkins and K. L. Myers, A common knowledge representation for plan generation and reactive execution, Journal of Logic and Computation **5**(6), pp. 731-761, 1995.