Colbert User Manual

Software version 8.0a (Saphira/Aria)
September 2001

©Kurt Konolige
SRI International
konolige@ai.sri.com
http://www.ai.sri.com/~konolige

1 Introduction

This tutorial describes the Colbert implementation in Saphira. It gives details on Colbert files, Colbert commands, control structures, and signals, and shows how to interface C/C++ variables and functions to Colbert.

The reader should be familiar with C and C++ syntax and semantics.

The files `colbert/direct.act`, `colbert/bump.act`, `tutor/movit`, and `tutor/loadable` are used as examples throughout the tutorial.

A useful overview of the Colbert system can be found in the Colbert paper.

2 Interacting with Colbert

Colbert is a programming language for robots. Its primary purpose is as an executive in a more complex 3-tier robot control architecture called Saphira/Aria (see Figure 2-1). In particular, Colbert can issue motion commands, can sequence robot actions, and can execute complex hierarchical control strategies using Saphira's repertoire of sensing and control routines.

Colbert is an executable language, which means that Colbert statements can be directly executed by an interpreter. The primary interaction most users will have with the robot is by issuing Colbert commands. These commands can load files, define robot activities, and start and control robot programs. They also enable the user to look at the state of the executing Saphira system.

Robot control strategies are written in Colbert as *activities*. An activity is a schema that contains Colbert commands and control structures. Schemas can be instantiated and run by the Colbert interpreter, and can issue the full ranges of Colbert commands.

Colbert is *extensible* in the sense that internal Saphira/Aria objects and functions can be made available through the Colbert interface. This includes new objects and functions defined by the user, as well as any of the pre-defined Saphira/Aria routines. Thus, a typical way to write and debug robot programs is to write them in C++, and to write companion Colbert activities that interface to these programs, and allow you to start, stop, and query them through the command interpreter.

Since it uses text-based commands, Colbert is also useful as a means of communicating and controlling a robot via a remote net-based interface. We won't go into the details of such an application in this manual, but the facility is available.

2.1 Colbert Interaction Window

Most of the time, users will issue commands to the robot through the Colbert interaction window. When Saphira starts up and loads its user interface, it creates a main window. The three parts of the window are a graphical display of the robot geometry (the Local Perceptual Space), a list of parameters about the robot and its communication on the left, and a Colbert interaction window on the bottom (see Figure 2-2).

Within the interaction window, users type Colbert commands. Each command is executed after the carriage-return, unless it is continued onto the next line with a trailing backslash ("\"). Any results from the command are printed in the same window. So, users can query the value of internal Saphira/Aria variables that have been made available to Colbert (Section 8).

All commands issued from the command line are done in `noblock` mode (see Section 7). For example, if you asked the robot to move forward one meter (`move(1000)`), then the command returns immediately, even though it takes the robot some time to move the required distance.

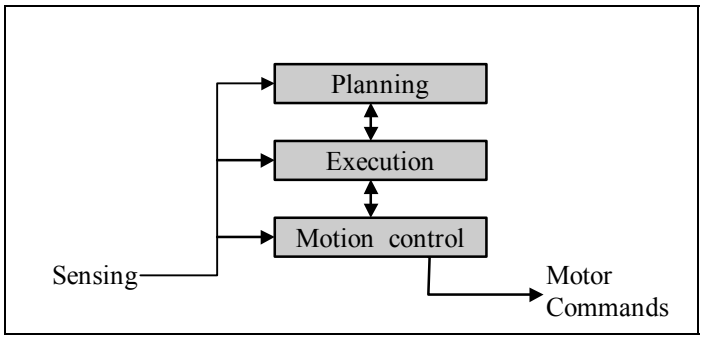

Figure 2-1 Schematic of the Saphira robot control architecture. The Colbert executive controls the action of the system.

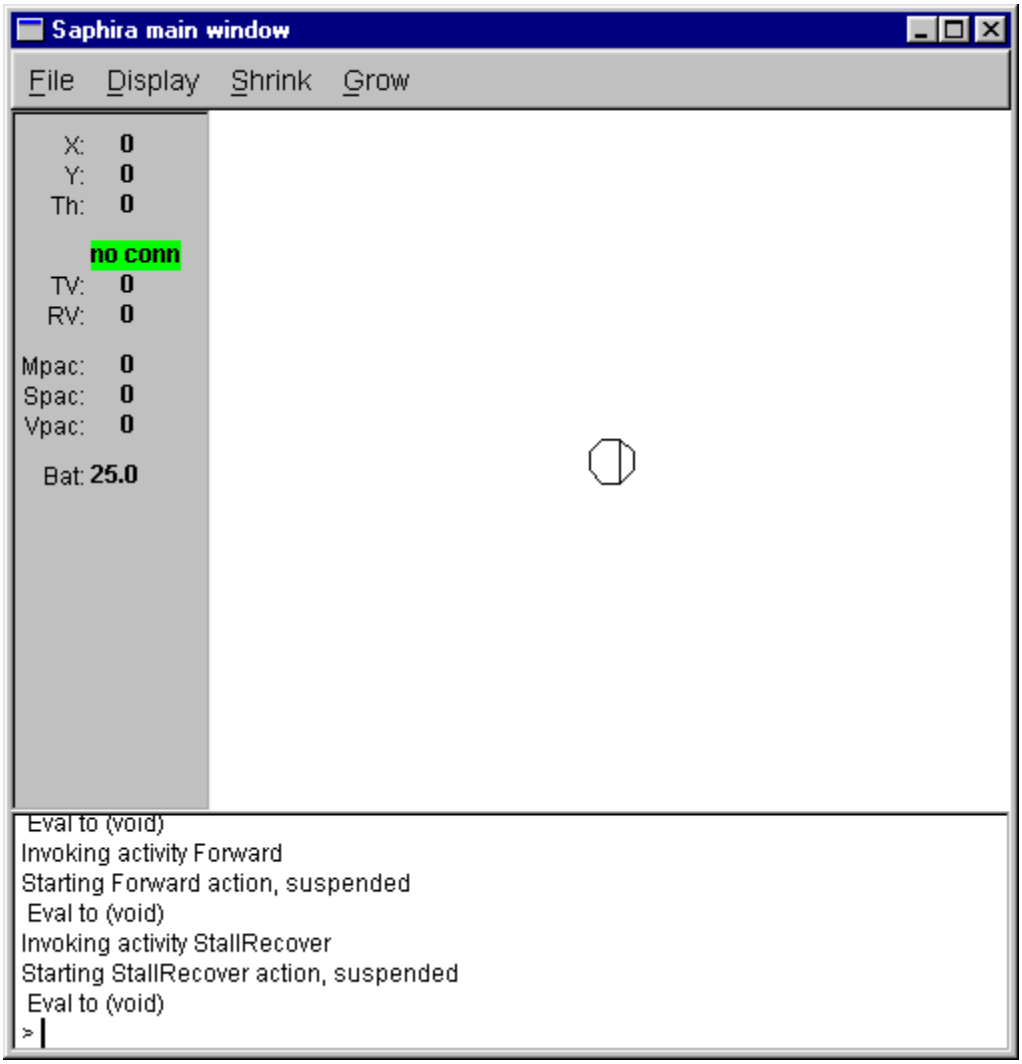

Figure 2-2 Saphira main window. The Colbert interaction window is at the bottom.

In addition to the results of commands, text from `sfMessage` commands is printed in the Colbert interaction window, if it exists. Tracing information from running activities is also printed here.

2.2 Colbert files

Colbert files, also called *activity files*, are text files containing Colbert statements. Usually the statements define a set of Colbert activities and global variables. They can also contain arbitrary Colbert commands, for example to load a file, or start an activity.

When Saphira first starts up, it looks for the special file `startup.act` in the `colbert/` directory. If present, it loads this file. The default `startup.act` loads the FLTK windowing system used by Saphira. For changing the default, see Section 10.1.

2.3 A Sample Colbert File: `bump.act`

The file `colbert/bump.act` is a sample file containing several activities, and associated Colbert commands. It illustrates the kind of file that will typically be written for controlling the robot.

The first part of the file, shown below, has some comments (in either C++ or C style), along with definitions of some global variables. All the variables in Colbert exist within a single name space, like C. Once defined, a variable cannot be redefined to be a different type. Note here the very simple Colbert

syntax, which is a subset of C --- it isn't possible to define and set a global variable in the same statement, it requires two statements.

```
//
// bump.act -- simple recovery from stall
//

//
// Saphira 8.x
// Kurt Konolige 2001
//

int MOVEVAL;
MOVEVAL = 250;
int BACKVAL;
BACKVAL = 200;
int TURNDEG;
TURNDEG = 40;
```

The interesting part of a Colbert file is usually the definition of *activities*. Activities are schemas for controlling the robot, with a finite-state machine semantics. For more information about activities, please see the paper on Colbert.

```
//
// activity to get out of a jam by turning and moving forward
// assumes we've already moved backwards
//

act getout(int dist, int turnDeg)
{
  if (sfStalledMotor(sfLEFT))
    turnDeg = -turnDeg;
  turn (turnDeg) timeout 10;// just in case, we timeout
  move(dist) timeout 10;
  actions;                  // resume behavioral actions
  succeed;
}
```

[omitted text, including definition of the `bng` activity]

```
start bng;
trace bng;
```

At the very end of the file, there are two Colbert commands. The first, `start bng`, starts up one of the activities in the file. Like Colbert commands issued in the interaction window, this one automatically is issued without blocking (Section 7), so execution of commands continues from the file.

The last command, `trace bng`, sets tracing for the `bng` activity. Whenever bng is active, it will start printing messages about the current line being executed (Section 11).

3 Data Types and Operators

Colbert's syntax is derived from C. But, it is not intended to be a replacement for C, and it implements only a subset of the C language. In addition, it contains some constructs that are useful for robot motion control, but are not strictly C syntax.

Colbert programs are not meant to be complex or computationally intensive. They give the user high-level control of the processes that operate the robot. Colbert is a flexible means of starting and stopping various processes that are needed in complex control tasks, and allowing the user to monitor, interact, and intervene in controlling the robot.

3.1 Basic Data Types

There are four basic data types in Colbert:

1. `int` integers
2. `float` single-precision floating point
3. `string` strings
4. `void` void pointers

These are the only basic data types that can be defined from within Colbert itself. Note the changes from standard C. There is no `double` floating-point type or `char` type; strings are a basic type. Void is used for declaring general pointers, which can point to arbitrary data types.

Another change from C is that there are no modifiers for basic types, such as `short`, `volatile`, `long`, `unsigned`, `static`, etc.

Integers and floating-point number are input with the standard C syntax, e.g.

```
3.41459     // floating point number
34          // integer, base 10
0x35        // integer, hexadecimal
```

Strings are input using standard C syntax for strings. Special characters such as "\n" and "\t" are defined, as well as the escape sequences "\0xx" for a single character.

3.2 Reference and Dereference Operators

The reference operator (&) and dereference operator (*) exist, as in C.

3.3 Assignment Operators

Only the standard assignment operator, "=", is defined in Colbert. No compound assignment operators are defined, e.g., "*=" or "+=".

3.4 Arrays

Arrays and array references are not supported in Colbert. In particular, you can't reference elements of a string data type from Colbert. However, string operations can be imported into Colbert by making C string functions available in Colbert.

3.5 Casting

Colbert does not support explicit casting operations. Implicit casting is performed in arguments to operators and functions.

3.6 Arithmetic, Logical, and Comparison Operators

The following arithmetic and logical operators are defined, with precedence as in C. The operators support implicit casting of types, e.g., adding a float and an int promotes the int to a float, and returns a float.

1. Arithmetic operators: `+ ++ - (unary) - (binary) -- * / %`
2. Logical operators: `| & ! ~`
3. Comparison operators: `< <= > >= || && == !=`

3.7 Structures

Structures cannot be defined within Colbert itself. However, Colbert supports creation and access of structures that have been defined in C or C++, and made available to Colbert (Section 9). With the advent of Saphira 8.x, structures are not used much, since their functionality has been replaced by objects. Unfortunately, the syntax and semantics of objects in C++ is complicated enough to preclude a direct Colbert interface. However, it is possible to interface static C++ functions to Colbert, and thus expose much of the functionality of the object-oriented approach of Saphira and Aria.

Assume that `mystruct` is a structure defined within a C/C++ file, and imported to Colbert. The structure has the following definition in the C/C++ file:

```
struct mystruct
{
  int s_int;
  float s_float;
};
```

Then, variables with a type `mystruct` can be defined in Colbert:

```
>mystruct a;
a declared
> a.s_int = 3;
a.s_int = 3
> a.s_float = 4.5;
a.s_float = 4.5
> mystruct *b;
b declared
> b = &a;
b = 0x3a0c45
> b->s_int;
b->s_int = 3
```

The operators "`.`" And "`->`" are defined in Colbert.

4 Standard C Commands

Colbert commands are used to control the operation of the Colbert executive and the robot. All Colbert commands can be issued directly to the command interpreter, either in the Colbert window, or at the top level of a Colbert file. They may also appear within Colbert activities.

This section defines Colbert commands that are equivalent to common C commands.

4.1 Declaring Global Variables

Global variables are declared using the basic data types and dereference operator, as in C. However, you cannot define a variable and set it in the same statement. Here is how some variables are defined and set in Colbert:

```
> int a;
a declared
> a = 4;
a = 4
> String b;
b declared;
> b = "abc";
b = "abc"
> int *c;
c declared
> c = &a;
c = 0x1345f650      // returns the address of the variable a
> int x = 4;  // syntax error, cannot set and define
*** Parsing error at token "="
```

A global variable, once defined, cannot be redefined. It exists for as long as the current Saphira application exists. Since there is only a single namespace, care must be taken to name global variables so they won't interfere with others that may be already defined. Any complex set of Colbert programs should have a naming convention in place to assure this.

4.2 C Expressions

Any C expression in Colbert's language can be evaluated as a command, and the result returned. Here are some examples:

```
> int a;
a declare;
> a = 43;
a = 43
> a + -12;          // this is a C expression
Eval to (int) 31    // return value type is given
```

4.3 Function Evaluation

C-type functions cannot be defined in Colbert; the main functional form is an *activity* (Section 6). However, C or C++ functions that are present in files loaded into Saphira can be made available to Colbert, via the extension interface (Section 7.4). These functions can be evaluated, using the standard C syntax. For example, the function

```
int myfn(int a);
```

is defined in `tutor/loadable/testload.cpp`, and made available in Colbert. Then, we can issue the function evaluation:

```
> myfn(1000);
Eval to (int) 1002   // result of evaluation
```

4.4 Assignment

Assignment is done with the "=" symbol only. The left-hand side of an assignment operation must be an lvalue. An lvalue is defined recursively:

1. Variables are lvalues
2. Struct members are lvalues
3. A dereference of an lvalue is an lvalue

5 Robot Motion Commands

Special commands in Colbert are available to move the robot. There are two types: *direct action* commands, which result in direct control over the robot; and *behavioral action* commands, which are actions that are combined as a set of behaviors to control the robot. The Saphira Tutorial on Actions is a useful document for understanding this section.

One peculiarity of the motion commands is that they can be issued in several modes. The modes are indicated by optional parameters to the command. For example, a motion command such as `move()` will normally cause the Colbert executive to wait for completion of the command before proceeding with the next command. But this action can be changed with the `noblock` parameter, which issues the move and then goes immediately to the next statement.

5.1 Direct Action Commands

These are the direct action commands:

Goal Movement	Continued Movement
`move(int mm)`	`speed(mm/sec)`
`turnto(int deg)`	`rotate(mm/sec)`
`turn(int deg)`	`stop`

The main distinction between the two types of actions, from Colbert's point of view, is whether they induce a goal movement or not. The actions in the first column, the *goal movement* actions, normally cause the Colbert executive to halt until their execution finishes. The actions in the second column, the *continued movement* actions, return immediately and continue execution, although their effects may last for an arbitrary amount of time.

Another distinction is whether the command causes a translation or rotational motion. The two types of motion are independent, so that a `move()` command can be active simultaneously with a `rotate()` command, for example (see the Saphira Tutorial on Actions). The `stop` command, which takes no arguments, stops both translational and rotational movement. To stop just translation, issue a `move(0)` command; for rotation, `turn(0)`.

All of these commands are *interruptible*, that is, issuing another translation command while a `move()` is active will halt the `move()`.

`move(int mm)` moves the robot forward (positive `mm`) or backwards (negative `mm`) a distance of |`mm`| millimeters.

`turnto(int deg)` turns the robot to point in the direction `deg`. This direction is according to the robot's global coordinate system.

`turn(int deg)` turns the robot incrementally an angle |`deg`|. This direction is counter-clockwise for positive `deg`, and clockwise for negative `deg`.

`speed(int mmsec)` sets a velocity setpoint for the robot to |`mmsec`| (in mm/sec). The velocity is forward if `mmsec` is positive, otherwise it is backwards. The robot will continue to move at this velocity until another translational command is issued.

`rotate(int degsec)` sets a rotational velocity setpoint for the robot to |`degsec`| (in deg/sec). The velocity is counterclockwise if `degsec` is positive, otherwise it is clockwise. The robot will continue to move at this velocity until another rotational command is issued.

`stop` stops all robot motion, canceling any translational or rotational commands.

5.2 Direct Action Options

Several options are available to change the default behavior of the direct action commands. The general form of a direct action command is:

```
command(arg) [noblock] [timeout n]
```

The options can be in any order. `noblock` causes the Colbert executive to continue executing the next statement immediately, without waiting for the movement command to finish. This option only makes sense on the movement goal commands (`move`, `turn`, `turnto`), since the others are effectively issued in noblock mode by default.

Movement commands issued to the Colbert interpreter in the interaction window, or at the top level of a file, are always issued in `noblock` mode.

The optional `timeout` argument specifies a maximum number of cycles for execution of the command. Each cycle is 100 ms in the current version of Saphira. `timeout` takes a single numeric argument, which must be a C integer; general C expressions are not allowed. Again, this option only makes sense for the movement goal commands.

`noblock` and `timeout` options can be issued together.

5.3 Behavioral Action Interface

Behavioral actions are defined as subclasses of `ArAction` in C++ code, and are made available to Colbert when they are loaded and their schema is added to the behavioral action schema list with `sfAddEvalAction`. For example, in the file `tutor/movit/movit.cpp`, the `SfMovitAction` action is added to the schema list by creating its schema instance:

```
sfAddEvalAction("Movit", (void *)SfMovitAction::invoke,
                2, sfINT, sfINT);
```

The `SfMovitAction` schema is now indexed under the schema name `Movit`, and Colbert commands will refer to this name.

Parameters to a behavioral action are specified within the invoke() function of the class. Here is the invoke() function for the `SfMovitAction` class:

```
SfMovitAction *
SfMovitAction::invoke(int distance, int heading)
{
  return new SfMovitAction(distance, heading);
}
```

The general format of the `invoke()` function is the same for all behavioral actions. It must be defined as a static member function. It can take up to seven arguments, and each should be a Colbert type (int, float, char *, or void *). Note that the `invoke()` function returns a new instance of the action. Although here the `invoke()` function and the constructor both have the same arguments, this is not necessary in general. In fact, to interface to an existing `ArAction` class that has non-Colbert arguments, simply define a subclass with only the `invoke()` function, containing just Colbert arguments.

The body of the behavioral action is contained in the `run()` function of the class. Within the body, various object variables are available for computation, including variables where the arguments have been stored by the constructor.

At every Saphira cycle (100 ms), an active behavioral action will have its `run()` function evaluated. The result of the behavioral action is specified by returning an `ArActionDesired` object. Details on how to set and return this object are given in the Saphira Tutorial on Actions.

Like direct actions, behavioral actions can be either goal-directed movements (e.g., go to a point) or continued movements (e.g., wander). Goal-directed behavioral actions should deactivate themselves when they finish. In SfMovitAction, this occurs when the robot has gone the required distance:

```
if (gone >= myDistance)
{
  sfMessage("Finished Movit");
```

```
    deactivate();            // turn off when done
    return NULL;
  }
```

The `deactivate()` function is called from within the `run()` function, to turn the action off. Note that `run()` returns NULL in this case, to indicate that it is not controlling the robot.

5.4 Invoking Behavioral Actions

The `start` command is used to invoke a behavioral action. The general form for this command is:

```
start <schema_name>[(arg1, arg2, ..., argn)]
              [noblock]
              [priority k]
              [timeout n]
              [iname <instance_name>]
              [suspend];
```

The `start` command starts a new instance of the behavioral action, or modifies an instance that is already present in the list of behavioral action instances. The instance name of the action is `<schema_name>`, unless the `iname` modifier is present, in which case it is `<instance_name>`.

A new instance is added to the behavioral action list if its instance name does not match the instance name of any other action on the list. If it does match, then Colbert checks to see whether the current instance is still running. If it is, then an error is issued. Otherwise, the current instance is removed, and a new one is started.

All behavioral actions will block Colbert execution until they are completed or signaled to finish. They will complete internally when the function `deactivate()` is called within the run() function. Externally, they can be signaled by activities (Section 7.4), or terminated by the optional `timeout` parameter. As with direct actions, a behavioral action can be given a maximum time to run, at which point it will be terminated by the Colbert executive. The time is specified in Saphira cycles (100 ms).

The optional parameter `noblock` starts the behavioral action in nonblocking mode. In this mode, the Colbert executive continues to execute immediately with the next statement in the activity, while the action continues to run in parallel. At the top level of a Colbert file, and within the Colbert interpreter window, all behavioral action commands automatically are issued in `noblock` form.

The priority of the behavioral action can be specified with the optional `priority` parameter. See the Saphira Tutorial on Actions for a description of the effects of priorities on behavioral actions. The default priority for an action is 0, that is, the lowest possible priority.

The keyword `suspend` starts the behavioral action in a suspended state. The action is present in the current action list, but it is not active, and does not contribute to the behavior of the robot. When suspended, its `run()` function is not called. Starting an action in the suspended state is useful if you want to have the action present for signaling, but don't yet want it to contribute to the overall behavior of the robot.

The order of the optional parameters is arbitrary.

5.5 Suspending Behavioral Actions

Behavioral actions and direct actions conflict with each other: it is not possible to execute both simultaneously. Direct actions have precedence over behavioral actions, so that if a direct action is executed, it will automatically turn off the behavioral action cycle.

The behavioral action cycle will stay off until it is explicitly turned back on (there is also a timeout mechanism; see `ArRobot::setDirectMotionPrecedenceTime()`). To allow behavioral actions, use the Colbert command `behaviors`, or the C++ function:

```
SfROBOT->clearDirectMotion()   .
```

6 Activity Definition

Activities are routines for controlling robot behaviors. Within an activity, it is possible to sequence robot actions, to start and stop subactivities, and in general to issue commands under programmed control.

Activities are *not* C or C++ functions. Their semantics is different, being based on concurrent finite-state machines. The paper on Colbert is a useful reference for the semantics of activities.

6.1 Activity Schemas

An activity is defined by specifying a *schema*, an activity body and its associated parameters. Examples of activity schema definitions are in the `colbert/direct.act` and `colbert/bump.act` files.

The general form of an activity schema definition is:

```
act <schema_name>[(<param1>, <param2>, ..., <paramN>)]
   {
      <declare_stmts>
      [update { <update_stmts> }]
      <activity_stmts>
   }
```

The parameter list is optional; if it is not included, the activity takes no arguments. The parameters are specified using the basic data types given in Section 3.1: `int`, `float`, `string`, or `void`, and the dereference operator "*". For example, the following activity schema takes two arguments:

```
act aa(int p1, float *p2)
   {
     sfMessage("%d %f", p1, *p2);
   }
```

This activity, when invoked, will just print out the values of its two arguments. Note that the second argument is a pointer to a floating-point number.

Arbitrary data can be passed in to an activity by the use of `void *` pointers. For example, Colbert cannot represent C++ objects such as artifacts. So, to pass an `SfPoint` object as an argument, specify the argument as a `void *` pointer.

The body of the activity has three distinct parts. In the declarations, any variables that are to be used in the activity are declared. If there are no variables, then this section would be empty. All variables are local to the activity. If there is a conflict between the name of a local variable and a global variable, the local one takes precedence.

The second part, the update part, is optional, and usually not present. The update statements are restricted to function and assignment statements. These statements are evaluated on every Saphira cycle that the activity is active, that is, is not in a completed state (see Section 7.1). Their purpose is to allow the calculation of information that may be important for the activity. For example, suppose the activity wants to check, in a number of places, whether there is a motor stall or not. Something like the following would be appropriate:

```
act aa()
   {
     int stalled;
    update
       { stalled = sfStalledMotor(sfLEFT) || sfStalledMotor(sfRIGHT); }
     while (1)
     {
       if (stalled)
        ...
```

```
    }
}
```

On every Saphira cycle, the value of `stalled` is computed from the `sfStalledMotor` functions. This value is then available within the rest of the activity.

The final part of the body contains activity statements that are executed as a finite-state machine by the Colbert executive. These statements are discussed in the next few subsections.

6.2 Activity Statements

The activity section of the body of an activity schema is composed of *statements*. Each statement is a Colbert command, such as assignment or movement command, or a control structure (`if` , `while`, or `goto`), or a label. Several special Colbert commands are available for signaling and other Colbert-specific actions.

6.3 Control Structures

Control structures change the flow of control within the body of the activity.

There is one conditional control structure, `if`. The syntax is the same as its C counterpart. This example is from `colbert/bump.act`:

```
if (sfStalledMotor(sfLEFT))
   turnDeg = -TURNDEG;
else
   turnDeg = TURNDEG;
```

Looping is performed by the `while` control structure. Its syntax is the same as its C counterpart. This example is the patrol activity from `colbert/direct.act`:

```
act patrol(int a)       /* go back and forth 'a' times */
   {
     while (a)
      {
        a = a-1;
        move(1000);
        turnto(0);
        move(1000);
        turnto(180);
      }
   }
```

The `goto` command changes the execution focus to a particular label within the body of the activity. It is an error to specify a `goto` command to a label that does not exist, and Colbert will catch this on the definition of the activity.

The example below shows how `goto` can be used to form a loop:

```
act patrol(int a)       /* go back and forth 'a' times */
   {
   begin:
     if (a <= 0) goto done;
     a = a-1;
     move(1000);
     turnto(0);
     move(1000);
     turnto(180);
     goto begin;
```

```
done:
}
```

6.4 Activity Labels

[This section is out of date]

Labels can be contained only within the activity statement portion of an activity. Their syntax is the same as in C, and they are targets for the `goto` command.

Several special labels are defined as the targets of external signals:

`OnInterrupt:` this label is the target of the interrupt signal (see Section 7.4). When an interrupt signal is sent to the activity, it branches to this label, and executes any commands found after it. Usually these commands will halt the robot, or clean up in some way. Note that the activity only gets one cycle to perform these commands; any command that causes a wait in the Colbert executive (e.g., a `move()` command issued without `noblock`) suspends further processing, and the activity enters the interrupted state.

`OnResume:` this label is the target for a resume signal sent to the activity. Whenever the activity is resumed from an interrupt or suspension, processing will start after this label. If the label doesn't exist, then processing on a resume starts from the first statement of the main body.

6.5 Wait Points

Coordination among activities often involves an activity waiting a certain amount of time, or waiting for a condition to become true. There are two Colbert activity statements for waiting.

```
wait <n>;
```

This command waits <n> Saphira cycles (100 ms), where <n> is an integer. Note that a specific integer must be given here, not a general C expression.

To wait until a condition is achieved, use:

```
waitfor <cond> [timeout <n>];
```

Here `<cond>` is a Colbert expression that is evaluated on every cycle. If it is non-zero, then the `waitfor` finishes and processing proceeds with the next statement.

An optional timeout parameter can be given to `waitfor` statements. If the `waitfor` does not finish before the given number of cycles, then it is terminated and processing proceeds with the next statement.

6.6 Termination Commands

An activity is finished when control reaches the end of the activity. For example, in the `patrol` activity above, when the variable `a` reaches 0, the `while` loop terminates and control reaches the end of the activity. At this point the activity has finished, and by default its termination state is `success` (see Section 7.1 for a description of activity states).

Activities do not return values, so there is no return statement corresponding to the `return` of C functions. It is possible to terminate an activity in ways other than by completion, however. There are two commands that terminate the activity immediately:

```
fail;       // terminate with failure status
succeed;    // terminate with success status
```

6.7 Redefining Activities

Activity schemas can be redefined, by issuing another `act` definition. The new definition supersedes the old one for all future instances of the activity. If there are any currently executing instances, they continue to execute with the old definition. So, if an activity file is edited and reloaded, any activity instances must be purged before the new definitions will take effect.

7 Invoking Activities

Colbert maintains a set of activity instantiations on an *activity list*. An activity instance is placed on the list using the `start` command (Section 7.2 below). For every activity on the list, the Colbert executive evaluates the activity once per Saphira cycle (100 ms). Depending on the state of the activity, the executive will evaluate some commands from the activity, then move on to the next. In this way, activities generate a sequence of commands to control the robot.

The activities on the activity list have both a parallel and a hierarchical structure. The nature of this structure is explained in Section 7.3. Activities can execute in parallel, and they can also wait for the execution of subactivities.

Activities can also receive and send *signals* to each other. Signals are used to change the state of an activity, or to remove an activity from the activity list (Section 7.4)

7.1 Activity State

Each activity on the list has an associated state, which is usually the current line at which execution is taking place. There are also some special states for activities:

`suspended` If an activity is suspended, it does not contribute to the behavior of the robot, and the Colbert executive skips processing it. A suspended activity can be restarted using the `resume` command (Section 7.4) or the `start` command (Section 7.2).

`timedout` The timed out state is similar to the suspended state, in that no execution takes place. It is the result of the activity using up its allotted time, given on invocation with the `timeout` parameter.

`success` This state is used by an activity to indicate that it has successfully completed its processing. No further execution takes place, unless the activity is signaled with a resumption signal.

`failure` Similar to success, but the activity has unsuccessfully completed its processing.

The state of all activities and behavioral actions is shown graphically in the activities window, if the windowing system has been loaded into Saphira (Figure 7-1). Any red entries are for behavioral actions. The state of the activity is shown in square brackets, and the dependency structure is indicated by indentation. In this case, the activity `aa` has started up two sub-activities, one of which has completed successfully.

Activity instances are shown in boldface red text. The `colbert/bump.act` files has been loaded, and the `bng` activity has started and is waiting for a motor stall at line 3 of the activity.

The state of an activity can be accessed in Colbert or C/C++ by using the `sfGetTaskState` function. This function takes an argument which is the instance name of the activity, and returns the state as an integer. For example, here is how to access the state of the bng activity:

```
> sfGetTaskState("bng");
Eval to (int) 12
> sfGetTaskState("Wander");
Eval to (int) 1
```

The return value is 12 for `bng` because the task is executing, and the state is the current line number + 9. States with integer values of 9 and below are reserved for special states of the activity. For example, the suspended state is state 1. Table 7-1 lists the special states, their numeric values, and variables that can be used to refer to them in Colbert or C/C++ code.

The state of an active behavioral action can be either `sfINIT` (0) or `sfSUCCESS` if it has been deactivated, which signals goal achievement. Behavioral actions can be suspended and removed, just like activities.

Several other functions access the state values of an activity.

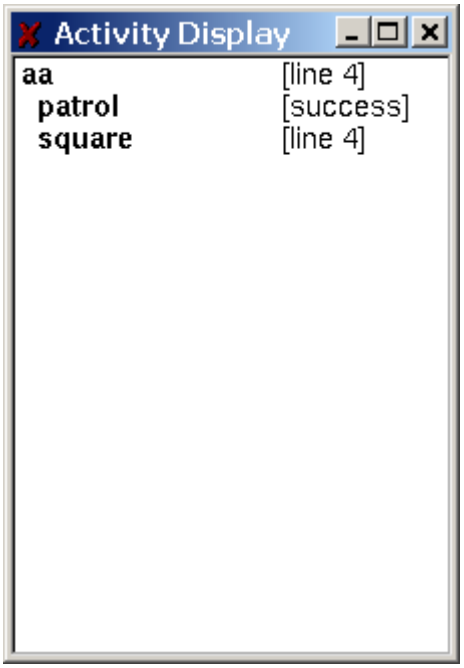

Figure 7-1 The Activities display window of the Saphira application.

```
sfTaskFinished(<instance_name>)
sfTaskSuspended(<instance_name>)
```

An activity is finished if its state is one of the completed states: timed out, success, or failure. If the named activity instance does not exist, `sfTaskFinished` will return true (1).

An activity is suspended if it is in either the suspended or interrupted state. If the named activity instance does not exist, `sfTaskSuspended` returns false (0).

7.2 Activity Invocation

The `start` command is used to invoke an activity, placing it on the activity list, or reactivating a suspended activity. The general form for this command is:

```
start <schema_name>[(arg1, arg2, ..., argn)]
            [noblock]
            [timeout n]
            [iname <instance_name>]
            [suspend];
```

The `start` command starts a new instance of the activity, or modifies an instance that is already

State	Numeric value	Symbolic value
suspended	1	SfSUSPEND
timedout	5	SfTIMEOUT
success	3	SfSUCCESS
failure	4	sfFAILURE

Table 7-1 Activity states and their numeric and symbolic values.

present in the list of activity instances. The instance name of the activity is `<schema_name>`, unless the `iname` modifier is present, in which case it is `<instance_name>`.

A new activity instance is added to the activity list if its instance name does not match the instance name of any other action on the list. If it does match, then the state of the current instance is checked. If it is finished, then it is removed from the activity list, and the new one is started. If it is not finished, then an error is issued, and no new activity is started.

All activities will block Colbert execution until they are completed or signaled to finish. They will complete internally when they fall through the last statement, or when explicitly terminated with one of the termination commands. Externally, they can be signaled by activities (Section 7.4), or terminated by the optional `timeout` parameter. As with actions, an activity can be given a maximum time to run, at which point it will be terminated by the Colbert executive. The time is specified in Saphira cycles (100 ms).

The optional parameter `noblock` starts the activity in nonblocking mode. In this mode, the Colbert executive continues to execute immediately with the next statement in the calling activity, while the called activity continues to run in parallel. At the top level of a Colbert file, and within the Colbert interpreter window, activity invocation commands are automatically are issued in `noblock` form.

The keyword `suspend` starts the activity in a suspended state. The activity instance is present in the current activity list, but it is not active, and does not contribute to the behavior of the robot. When suspended, its body is not executed by the Colbert executive. Starting an activity in the suspended state is useful if you want to have the activity present for signaling, but don't yet want it to contribute to the overall behavior of the robot.

The order of the optional parameters is arbitrary.

7.3 Activity Structure

Activity execution is controlled by the parallel and hierarchical structure of the activity list. Please see the Colbert paper for a technical description of this structure. Here, we describe the implementation and practical aspects of the structure.

All executing activities are held on the activity list. The activities form a hierarchy that consists of a set of trees (a *forest*). At the top level, each activity is the root of a tree. There can be many such activities; all of them execute in a round-robin fashion, with the Colbert executive evaluating them in their order on the list. The order is determined by their instance name: instance names with a lower alphabetic precedence are placed higher on the list, e.g., an activity name "`a`" appears before one name "`b`". Note that for efficiency reasons the activity window does not list top-level activities in this order, but rather in the order they were created.

Whenever a new activity is invoked from within another activity, it becomes a *subactivity* of its caller. This parent/child relationship forms a tree structure from each of the top-level activities. The tree structure of a top-level activity is indicated graphically in the activity window, by indenting the child activities below their parent. Activities at the same level of indentation are all children of the activity immediately above them at a lesser indentation (Figure 7-2).

If all of the subactivities in the tree are invoked in blocking mode (the default), then the tree is always linear (single-branching), and only the leaf node is executed by the Colbert executive; all of the other

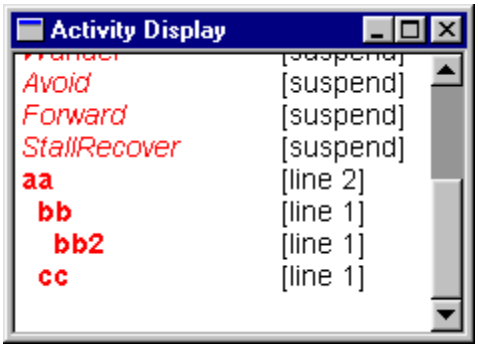

Figure 7-2 Hierarchical activity structure displayed in the activity window. `bb` and `cc` are children of `aa`; `bb2` is a child of `bb`.

activities are waiting for completion of subactivities. In this case, the top-level activity gives rise to only a single point of execution.

It is possible to execute several subactivities of a top-level activity tree in parallel. In this case, the tree can be more than single-branching, as in Figure 7-2. This tree was generated by the following commands to Colbert:

```
> act aa { start bb noblock; start ww iname cc; }
Defining aa
> act bb { start ww iname bb2; }
Defining bb
> act ww { waitfor 0; }    // this just waits indefinitely
Defining ww
> start aa
Invoking activity aa
```

Note the presence of the keyword `noblock` in activity `aa`. With this keyword, activity `bb` is started up, and processing of `aa` proceeds with the next statement, while `bb` continues executing. Thus, `noblock` starts a new line of execution, and makes possible a branching execution structure within the Colbert executive.

7.4 Signals

Signals allow activities to change the state of other activities. Typically, activities can be suspended or removed by a *monitoring* activity. For example, in the bump.act activity file, the activity bng is defined as:

```
act bng()
{
  while(1)
  {
    waitfor(sfStalledMotor(sfLEFT) || sfStalledMotor(sfRIGHT));
    remove getout;          // halt this activity if it's going
    stop();                 // stop the robot
    move (-BACKVAL) timeout 30;  // just in case, we timeout here
    start getout(MOVEVAL, TURNDEG) noblock;
  }
}
```

After waiting for a motor stall, `bng` removes the activity `getout` from the activities list. `getout` is an activity that turns the robot and moves forward. It is called by `bng` to move the robot away from an obstacle after it has backed up. But, during the `getout` motion, the robot may again encounter an obstacle, and so `bng` first removes any instance of `getout` before issuing its commands.

Table 7-2 lists the signals that can be sent, and the corresponding Colbert and functional commands to send them.

Signals apply recursively to any sub-activities of a signaled activity. For example, if an activity is suspended, then all its dependent activities are also suspended. This is a good way to suspend or resume a group of activities, i.e., just start them all as dependents of a single activity.

Signal	Colbert command	C++ command
`remove`	`remove iname`	`sfRemoveTask(char *iname)`
`suspend`	`suspend iname [n]`	`sfSuspendTask(char *iname, int time)`
`interrupt`	`interrupt iname`	`sfInterruptTask(char *iname)`
`resume`	`resume iname`	`sfResumeTask(char *iname)`
`success`	`succeed iname`	`sfSucceedTask(char *iname)`
`failure`	`fail iname`	`sfFailTask(char *iname)`

Table 7-2 Activity signals.

8 State Reflection Functions

9 Extensions: Interfacing to C/C++

It is convenient to extend the Colbert base by adding interfaces to C/C++ functions, variables and objects, as well as to new behavioral action schemas. Colbert allows such extensions, and we discuss them in this section. Please see the `tutor/movit` files for an example of adding behavioral actions, and `tutor/loadable` for functions and variables.

The Colbert interface is always extended by calling one of the interface functions from C/C++ code. The four interface functions are `sfAddEvalConst`, `sfAddEvalVar`, `sfAddEvalFn`, and `sfAddEvalAction`.

One difficulty with interfacing occurs with C++ objects. The syntactic abilities of Colbert are limited, and the syntax and semantics of C++ objects are too difficult to deal with. Instead, objects can be passed to Colbert cast as `void*` pointers. The interface functions can recast them to the correct object type, and access their member functions and data.

Colbert has a facility for dealing with structures, but it is not used very much, and we will not describe it here.

9.1 Identifiers

Colbert identifiers are alphanumeric strings, and stand for functions, constants, variables, behavioral actions, and activities. There are two global namespaces: one for functions, constants and variables, and one for behavioral actions and activities. A single identifier can be used for a variable and an activity, for example; but it can't be used for both a variable and a function, or an action and an activity.

9.2 Constants

Constants are data that are defined and then never change. Constant data are added using the `sfAddEvalConst` function.

```
sfAddEvalConst(<string-id>, <type>, <value>)
```

Here `string-id` is an alphanumeric string, `type` is a Colbert type, and `value` is the value of the constant. For example, to add a new constant named `sfFIRST`, with value (int)1, use:

```
sfAddEvalConst("sfFIRST", sfINT, 1)
```

9.3 Variables

Variables are data that are defined and can be set. Variables are added using the `sfAddEvalVar` function.

```
sfAddEvalVar(<string-id>, <type>, (fvalue *)<value-ptr>)
```

Here `string-id` is an alphanumeric string, `type` is a Colbert type, and `value-ptr` is a pointer to the variable. This pointer must be cast as an `fvalue*`. For example, to add a new variable named `myvar`, which is tied to the global integer `gvar`, use:

```
sfAddEvalVar("myvar", sfINT, (fvalue *)&gvar)   .
```

After evaluating this expression, the Colbert symbol `myvar` will be tied to the C/C++ global variable `gvar`. Whenever `myvar` is accessed in Colbert, it will return the current value of `gvar`; and when `myvar` is set in Colbert, it will set the value of `gvar`.

Both C and C++ variables can be interfaced to Colbert.

9.4 Functions

C and C++ global functions are interfaced to Colbert using the `sfAddEvalFn` function.

```
sfAddEvalFn(<string-id>, (void *)<fn>, <rtype>, <numargs>,
   <type1>, <type2>, …)
```

Here `string-id` is an alphanumeric string, `fn` is a global function, `rtype` is a Colbert type, `numargs` is an integer, and `typeN` are Colbert types. The cast of the function to type `void*` is mandatory. `rtype` is the return type of the function; `numargs` is the number of arguments to the function, and `typeN` are the argument types, in order.

For example, to add a new function named `myfn`, which is tied to the global function `gfn`, use:

```
sfAddEvalFn("myfn", (void *)gfn, sfINT, 1, sfINT) .
```

After evaluating this expression, the Colbert symbol `myfn` will be tied to the C/C++ global function `gfn`. The function `gfn` can be invoked from Colbert using normal function syntax, e.g.

```
int a;
int b;
a = 3;
b = myfn (a);
```

Note that only global functions can be accessed via Colbert. In C++, this means only global and statically-defined functions are available, not standard object member functions. For example, there is no direct way to access the state reflection functions of an `ArRobot` object. Still, by using indirect methods, it is possible to evaluate member functions of objects. Here is a simple example: we want to return the X coordinate of the current robot (`SfROBOT`). First, we define a global function to do this:

```
float robotX()
{
  return (float)SfROBOT->getX();
}
```

Note that we have changed the type of the return to `float` rather than `double`, since Colbert can only use `floats`. Next, we add this global function to Colbert:

```
sfAddEvalFn("robotX", (void *)robotX, sfFLOAT, 0) .
```

Now, the value of the current robot's X position can be accessed from Colbert, using the `robotX` function.

This example assumed there was a global variable or function, `SfROBOT`, to access the current robot. In fact, it is easy to make `robotX` return the X coordinate of a robot object that is passed in as an argument. To do this, we cast the robot object as a `void*` pointer.

```
float robotX(void *robot)
{
  return (float)((ArRobot *)robot)->getX();
}
```

Now we add this global function to Colbert:

```
sfAddEvalFn("robotX", (void *)robotX, sfFLOAT, 1, sfPTR) .
```

The function `robotX` will take a pointer to a robot object, and return the X coordinate of that robot. Of course, we still need more functions to generate these pointers in Colbert as void* pointers (`sfPTR` type). The file `tutor/loadable/testload.cpp` has an example of access an `SfPoint` object using Colbert functions.

9.5 Behavioral Actions

Behavioral actions (Aria `ArAction` classes) are interfaced to Colbert using the `sfAddEvalAction` function.

```
sfAddEvalAction(<string-id>, (void *)<action>::invoke(),
   <numargs>, <type1>, <type2>, …)
```

Here `string-id` is an alphanumeric string, `action::invoke()` is a static invocation function of the action, `numargs` is an integer, and `typeN` are Colbert types. The case of the function to type `void*` is mandatory. `numargs` is the number of arguments to the function, and `typeN` are the argument types, in order.

For example, to add a new action named `myact`, which is tied to Aria action class `act`, use:

```
sfAddEvalAction("myact", (void *)act::invoke(), 1, sfINT) .
```

After evaluating this expression, the Colbert symbol `myact` will be tied to the C++ action class `act`. A new instance of the action can be invoked from Colbert using the start command (see Section 5.4).

10 Colbert Files

Colbert files, also called *activity files*, are text files containing Colbert statements. Usually the statements define a set of Colbert activities and global variables. They can also contain arbitrary Colbert commands, for example to load a file, or start an activity.

10.1 Special Files

Colbert files can be used to set up a Saphira system for a particular application, by loading shared object files and other Colbert activity files, and starting up a set of activities.

When Saphira first starts, it looks for the Colbert file `colbert/startup.act`, and if present, loads it. The startup file supplied with the Saphira distribution does three things:

1. Loads the FLTK windowing system, which starts up a graphical user interface.
2. Load the sonar processing functions.
3. Starts several basic actions, and leaves them in a suspended state.

You can customize Saphira by changing the startup.act file. For example, you may not want the basic actions to be started. Or, if Saphira is running on an embedded system, you may not want to load the FLTK window system. Finally, you can load other files that define more capabilities for Saphira.

10.2 File Structure

11 Miscellaneous Commands

Tracing