SAPHIRA TUTORIAL

Motion Control: Direct Control and Behavioral Actions
Software version 8.0 (Saphira/Aria)
September 2001

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

1 Introduction

This tutorial discusses methods for controlling a robot's motion using the interfaces provided by Saphira and Aria. Several examples of movement control are discussed; these examples can be run directly on the robot or the simulator.

The purpose of this tutorial is to illustrate basic motion control, relative to the robot's internal geometry. The internal geometry is generated by integrating the robot's motor encoders, which determine how far the wheels have traveled. Because of imperfections in the wheels, slippage on the floor, and other environmental factors, the internal geometry will only approximately reflect the actual geometry of the robot motion. For example, controlling the robot to move along a straight line causes it to move in a straight line according to the internal geometry, but its real motion will deviate from this ideal case, eventually curving and moving away from a straight line in the real world. The only way to correct these deviations is by *localizing* the robot through sensing of the environment. This subject is quite complex, and is discussed in other tutorials. Here we assume all motion is with respect to the internal geometry.

Examples mentioned in the tutorial are available in the `tutor/movit` directory.

Other relevant materials are the Saphira User's Manual, the Aria documentation, and the Colbert User's Manual.

2 Moving the Robot in Saphira

There are two control interfaces to robot motion that can be utilized in Saphira. Figure 2-1 diagrams the difference between these interfaces.

1. Direct control. In this mode, user programs communicate directly to the motor controller on the robot, and request movement along two independent motion channels, translation and rotation. This mode is generally thought of as a *monolithic* controller, because the control computation occurs inside a monolithic block, and the results are used directly to control robot motion.
2. Behavioral control. In this mode, motion is mediated by sets of *behavioral actions*, which are schemas for determining motion based on sensory input and internal state. The results of behavioral actions are combined by a *resolver* to produce motion. This method is an indirect or *partitioned* way of controlling robot motion, since the end result is made up of a contribution from many smaller modules.

Which interface is best depends on the application. Both interfaces are always present, and programs can switch between them, depending on the needs of the moment.

In general, direct control is preferable when the user's program performs a monolithic computation of the best motion for the robot to take. The output of the computation can be applied directly to the robot, using the direct motion commands.

Another case where direct control works well is in writing complex sequences of movements. For example, suppose you want to write a control sequence for getting the robot unstuck after bumping into something. A simple sequence would be to back the robot up a small amount (and check if it hit anything in backing up!), turn a bit, and then move forward, checking again if an obstacle is there. This sequence can best be written as a finite-state control in Colbert, where at each state the control issues a direct motion command such as backing up.

Behavioral control is more suited for applications that break down complex activities into small, modular packages. These packages typically combine sensing and action into a coherent *behavior*, hence

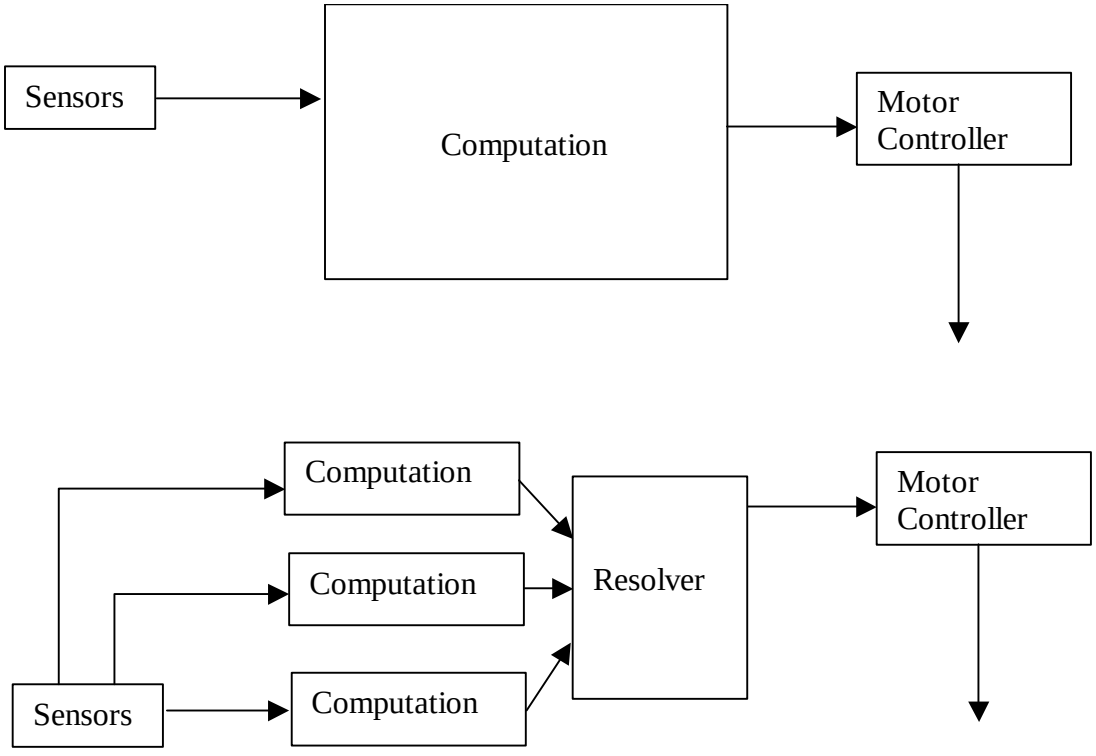

Figure 2-1 Two different ways of controlling robot motions. Top: a monolithic (direct) motion controller. Bottom: a partitioned (indirect) motion controller.

the name. For example, an *Avoid* behavior might slow the robot down and turn it away from an approaching obstacle. This simple behavior could be combined with others, for example, a *Goto* behavior that moved the robot towards a goal. By mixing behaviors in the right manner, programs can produce coherent activity.

In Saphira, behaviors are called *behavioral actions*, or simply *behaviors*. Sets of such actions, together with a resolver for mediating among them, comprise the behavioral control component of Saphira.

3 Direct Motion Control of the Wheels

Pioneer, PeopleBot and Amigobot robots are differential-drive robots with two wheels on an axis that is approximately through the center of the robot. Each wheel is coupled to an independently-driven motor, which also contains an encoder for feedback about the wheel motion. A microcontroller on the robot implements a control algorithm for turning the wheels in a goal-oriented manner. For example, to achieve a given velocity and heading, a *feedback controller* changes the motor power to move the velocity or heading to the desired value, and keep it there. The desired values are called *setpoints*. In typical digital controllers, the power value is recalculated hundreds of times a second, to achieve a tight tracking of the setpoints. Most bases provide such setpoints as part of their control electronics.

3.1 Direct Motion Commands

What are the choices for setpoints? There are two control channels: rotation and translation. For each of these channels, we can choose to have either a velocity or position setpoint. (Why not acceleration?) For example, we could choose to move the robot a distance d and turn it to an angle ϕ. Or, we could choose to control the robots translational velocity v and rotational velocity ω. Table 3.1 shows the possibilities for each of the control channels. A particular *control regime* consists of one choice from each column. You can also control the wheel velocity of each wheel independently, using the `SetTVel2` command.

The direct motion interface allows you to work in any control regime, and to switch control regimes at will, by calling the indicated functions. The lowercase functions are Colbert commands that can be invoked from Colbert activities, or the command-line interface. The capitalized functions are member functions of an `SfROBOT` object, the current robot. `SfROBOT` is a subclass of the Aria `ArRobot` class, and the direct motions functions are invoked as its member functions, e.g.,

```
sfROBOT->setVel(200) .
```

Note that the velocity and position control have very different effects on robot motion. Calling `setVel(100)`, for example, will start the robot moving until it attains a velocity of 100 mm/sec, and keep it moving at that speed. On the other hand, calling move`(100)` will cause the robot to move forward 100 mm from its current position, and then stop.

Rotational and translational velocities can be incremented, as well as set to a particular value.

Rotational position has two functions: one for absolute angular position within the internal coordinate system, and one for relative position. `setDeltaHeading(`ϕ`)` moves the angular setpoint ϕ degrees from its current position. `setHeading(`ϕ`)` moves the setpoint to the absolute position ϕ.

Maximum values for velocity during positional commands `move()` and `setHeading()` are available through the `setMaxTransVel()` and `setMaxRotVel()` functions. You can use these to make your robot move more slowly or more quickly during positioning commands.

All of these control functions are *interruptable*, that is, issuing a new translational command interrupts the current robot translational motion by resetting the setpoint. It may also switch regimes, e.g., if the

	Translation	Rotation
Velocity *mm/sec* *deg/sec*	`speed(v)` `setVel(int v)`	`rotate(v)` `setRotVel(int` ω`)`

Position
mm
deg
	`move(d)`
	`move(int d)`
		`turnto(`ϕ`)`
		`setHeading(int` ϕ`)`
		`turn(`ϕ`)`
		`setDeltaHeading(int` ϕ`)`

2 Wheels *mm/sec*	`setVel2(left, right)`	
Stop	`stop` `stop()`	

previous command was a positional one, and the new one is for velocity, then translational control is switched from position to velocity.

3.2 A Colbert Example

It's easy to use the direct motion commands from Colbert: there are special Colbert functions that map directly to direct motions (Table 3.1). In the tutorial directory `movit`, the activity file `movit.act` contains several examples of activities that exercise the direct motion commands. We'll look at one of these, the `patrol` activity.

We want the robot to patrol up and down between two goal points, repeating this activity a specified number of times. We'll use the direct motion commands of (1) turning to a heading, and (2) moving forward a given distance.

The simplest way to realize the patrol activity is as a perpetual `while` loop, in which the direct 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 3-1 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^o direction, then moves forward 1000 mm, then turns to the 0^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 direct motion commands 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. Direct motions 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.

3.3 Invoking the Patrol Activity

A Colbert activity can be invoked from another activity, or directly from the Colbert command line interface. For the patrol example, we first load the activity from the file `movit.act`, then invoke the activity using the `start` command. This is the sequence of commands to use:

```
cd ../tutor/movit;      // connect to the movit directory
load init.act;          // load the activity file
start patrol(10);       // this command starts up the activity
```

If you start up Saphira, and type in this sequence of commands, nothing will happen – you have to connect to a robot first, of course. Only when Saphira is connected to a robot will motion commands have any effect. Otherwise, the `patrol` activity will give an error at the first motion command, and suspend itself.

The complete form of the `start` command is:

```
start schema(arg1, arg2, …) [name instance-name]
                            [timeout ticks]
                            [priority pri]
                            [suspend]
                            [noblock]
```

schema is the name of the activity schema, as written in the activity file. There should be as many arguments as parameters in the schema. Colbert will perform simple type conversions, e.g., from floats to ints.

The optional arguments control various aspects of the activity. You can give this particular instance of the schema a unique name (otherwise it defaults to the schema name). This is useful if you want to have multiple instances of the same activity schema.

Activities can have an attached timer that monitors the duration of the activity. If the timer expires, the activity is suspended. The timer is initialized with the `timeout` argument, where *`ticks`* is the number of system cycles (100 ms) the activity is allowed to run.

An activity can be invoked in a suspended state, so that it is present and ready to run, but not currently active.

The `noblock` argument is useful when calling an activity from another activity. It lets the caller keep running, while splitting off the new activity to execute in parallel. The default behavior, when one activity calls another, is to wait for the called activity to complete before the caller continues with its program. From the command line, all activities are implicitly started in a noblock state, since they start running while the command line processor continues to accept input.

Once an activity is completed, it will still be present in the system, although it won't be active. To remove it completely, use the `remove` command, with the name of the activity. You can also interrupt the activity at any point by sending it a signal with the `interrupt` command; the `resume` command resumes it. All of these commands use the instance name of the activity, which is usually the same as the schema name.

4 Behavioral Control

An alternative to direct motion control is, of course, *indirect* control. Under an indirect control regime, complex actions are partitioned into smaller units, each of which suggests motions to perform (see Figure 2-1, bottom).

Behavioral actions are implemented in Aria. The smallest unit of behavior is called an *action*. Actions are implemented as objects, and action types (or *schemas*) are the object classes. There is one base class for actions, called `ArAction`, from which particular actions types are derived as subclasses. Examples of some action types can be found in `Aria/examples/actionExample.cpp`.

Behavioral actions can the same types of control values as direct actions: translational velocity and position, and rotational velocity and heading. But, since the outputs are meant to be combined, it is important that the current set of executing actions use the *same* control on each of the channels. For example, if one action attempts to control the heading of the robot, and another tries to control rotational velocity, there is no way to combine these outputs. It is up to the user to use a consistent control scheme across invoked actions. We recommend using translational velocity and rotational heading as the standard action outputs for most uses.

Action outputs are combined by a *resolver*. In most cases, the default resolver provides a good way to combine actions: it is based on priority classes (see below). In advanced programming, users can write their own resolvers; see the Aria documentation.

4.1 The Action Evaluation Cycle

The set of currently active actions is held on a list in the robot object `SfROBOT`. On every cycle (100 ms), each action object is evaluated to produce a translational and rotational output, along with a strength for each. The strength, which varies from 0 to 1, indicates how strongly the action prefers to have this motion executed. The output values for behavioral actions are described by a structure, `ArActionDesired` (see `Aria/include/ArActionDesired.h`).

Once the outputs of all current actions have been computed, they are given to a *resolver* to determine what the final output will be. There are many possible types of resolution strategies: averaging, winner-take-all, competition, etc. Users are free to define their own resolution strategies to fit particular application needs; these strategies are defined by subclass the Ar`Resolver` class. Aria's standard resolution strategy is a two-part resolution strategy (Ar`PriorityResolver`) that is the default.

In priority resolution, each action instance is given a priority number, which is a positive integer. Higher numbers indicate higher priority. Within each priority class (defined as those objects that have the same priority number), the action outputs are averaged according to their strengths. To determine the final motion, the priority classes are searched from highest to lowest. An activation energy is initialized to 1 for the highest level, and the level "uses up" the activation according to the strength of its strongest action, i.e., a strength of 1 would use up all the activation, while a strength of 0 would use up none. At the next level, the motion command is averaged in, but discounted by the activation energy remaining. Again, the level uses up activation based on the strength of its motion output.

Priority resolution allows high-priority levels to completely block lower levels when they are strongly active. But, it lets lower priority actions influence motion when the higher priority actions are only weakly activated. For example, a `GoTo` action at a lower priority would remain active, as long as a higher-priority `Avoid` action did not detect an obstacle and try to avoid it. The use of numeric activations allows a smooth transition between these behaviors.

[**NOT PRESENT – NEEDS UPDATING ***** An example of the tradeoff of behavioral actions is the `Avoid` and `Forward` actions (`ohandler/src/basic/actions.cpp`). The standard Saphira client starts up with suspended instances of these two actions. The `Avoid` instance has a priority of 20, and the `Forward` instance a priority of 10. If you enable them (by left-clicking in the Activities window), then the robot will move forward at a moderate speed, until it encounters an obstacle. At this point, the `Avoid` action will become dominant, and slow and turn the robot away from the obstacle. Then, the `Forward` action takes over again. Using just this simple tradeoff, the robot wanders around, without bumping into anything.

A similar behavior is implemented by the `Wander` action. Here, both moving forward and avoiding obstacles are part of a single behavioral action. The use of a single action of this kind can often result in more tightly coupled and purposeful motion, than the combination of separate actions.]

4.2 The Behavioral Action `SfMovitAction`

In this section we'll examine a behavioral action in detail. The `SfMovitAction` subclass is defined in `Saphira/tutor/movit/movit.cpp`. The purpose of the action is to turn the robot to a given heading, and move it forward a given distance. It's not a particularly useful action, but it will exercise the capabilities of the system.

Here is the subclass definition:

```
class SfMovitAction : public ArAction
{
 public:
  SFEXPORT SfMovitAction(int distance, int heading); // constructor
  virtual ~SfMovitAction() {};// nothing doing
  // this defines the action
  virtual ArActionDesired *run(ArActionDesired currentDesired);
  // if we need to reset distance to go
  SFEXPORT void reset() { gone = 0; ax = SfROBOT->getX();
                          ay = SfROBOT->getY(); }
  // interface to Colbert
  static SfMovitAction *invoke(int distance, int heading);

protected:
  ArActionDesired myDesired;  // what the action wants to do
  int myDistance, myHeading;  // parameters to the action
  int gone;                   // how far we've gone
  double ax, ay;       // old robot position
};
```

Although this might seem complicated, a lot of it is boilerplate to interface actions to the behavioral controller. The first three functions, `SfMovitAction`, `~SfMovitAction`, and `run`, are required. The code in `run()` will contain the actual body of the action, which is where all the "action" is.

The `Reset()` function is the only one not required. We've defined it here to reset the parameters of the action, as an illustration of how we can affect an action under program control. In this case, calling `Reset()` will zero out the distance the action thinks the robot has traveled.

Private variables hold information the action needs for its processing. Here, `ax` and `ay` hold the previous position of the robot, so we can accumulate the distance the robot has traveled, in the variable `gone`. `myDesired` is the output of the action, and `myDistance` and `myHeading` hold the parameters.

To interface actions to the Saphira/Colbert system, a special member function `invoke`() is defined statically. `invoke()` acts like a constructor, and should return an instance of the action, using its arguments to set up the action parameters.

4.3 The Action Constructor

The action constructor mints a new action instance. You can create actions instances directly using the `new` operator on the class, which invokes the constructor. But, action instances created this way will not be seen by Colbert and the Saphira activity executive. Within Saphira, the sfAddEvalAction is used to add the action class to the list of known actions, and then `sfStartTask()` can be used to start up instances of the action from C++ code, or the `start` operator from Colbert (Sections 4.5 and 4.6).

Here is the text of the `SfMovitAction` constructor. This constructor follows the pattern for all other constructors of action subclasses. The only particular change is the number and type of arguments used by the action.

```
SFEXPORT
sfMovitAction::sfMovitAction(char *name, bool instance)
    : ArAction("Movit")
{
```

```
  myDistance = distance;
  myHeading = heading;
  reset();
}
```

Note the constructor chaining performed by calling `ArAction("Movit")` in the prolog of the constructor. This chaining is mandatory to set up the action instance correctly. After that, the internal arguments are set by the constructor.

4.4 The Action Body

The action body is defined by the `run()` function. This function is called on every cycle that the action is active, and is responsible for determining the output of the action. The output results are always returned as a pointer to an `ArActionDesired` structure. Note that the `run()` function also takes an `ArActionDesired` argument as input. This input is set by the action executive to be the current state of outputs from (higher-priority) actions already evaluated; generally it is ignored for most actions.

Here is the body of the `SfMovitAction` action. Remember that this action is supposed to move the robot forward by its first argument, and turn it to the heading given by its second argument.

```
SFEXPORT ArActionDesired *
SfMovitAction::run(ArActionDesired d)
{
  // reset the actionDesired (must be done)
  myDesired.reset();

  // check the distance to be traveled
  double dx = ax - SfROBOT->getX();
  double dy = ay - SfROBOT->getY();
  ax = SfROBOT->getX();       // set new values
  ay = SfROBOT->getY();
  int ds = (int)sqrt(dx*dx + dy*dy);
  gone += (int)ds;
  sfMessage("Running Movit, gone %d", gone);

  if (gone >= myDistance)
    {
      sfMessage("Finished Movit");
      deactivate();           // turn off when done
      return NULL;
    }
  else
    {
      myDesired.setHeading(myHeading); // control the heading
      myDesired.setVel(200);  // moderate speed
    }
  return &myDesired;          // return the desired controls
}
```

The arguments to the action are present as variables of the action instance; in this case, they are `myDistance`, and `myHeading`. Other object variables, such as `ax` and `ay`, will persist over the lifetime of the object and can be used for extended computations over calls to the `run()` function.

The action body itself should not have a lot of computation, because it must share the 100 ms microtask execution cycle with many other processes. Here, we just add in the incremental change in the robot's position, using the information present in the current robot object `SfROBOT`. The variable `gone` holds the accumulated distance the robot has moved since the action was instantiated (or since the last `reset()` call).

To move the robot the desired distance, the distance traveled is checked against the given distance, and if it is not yet achieved, the action controls both heading and speed to achieve the goal. This action is all-or-none: it either tries to fully control the robot motion, or leaves it completely alone. For smoother integration with other actions, it would be useful to gradually decrease control over motion, say as the

robot comes near to the goal. Note that the default activation value on `setHeading` and `setVel` is 1.0, the maximum activation.

Once the desired distance has been traveled, the action deactivates itself. For *goal directed actions*, such as this one, it is important to call `deactivate()` once their task is done. Deactivation has two effects. First, it frees up the action executive from having to process the `run()` function any more. Second, it communicates to Saphira and Colbert that the action has completed. If there are other activities that are waiting for the action to complete, they can then proceed.

If an action does not want to control the movement of the robot, it can return a NULL pointer instead of the `ArActionDesired` pointer. This is done when the action finishes.

Actions that are continuing ones, rather than goal directed, will generally not deactivate themselves. For example, an action that keeps the robot from bumping into obstacles is a continuing action: it never achieves a goal position. Continuing actions are usually deactivated from outside the action itself, e.g., by the Colbert executive.

4.5 Interfacing the Action to Saphira/Colbert

There are two parts to creating the Saphira and Colbert interface to an action. First, the `invoke()` function must be defined statically, and serves as a function that can be called from Colbert, just as other functions made available with `sfAddEvalFn`. Invoke() takes arguments that are Colbert types (ints, floats, and void * types), and returns an instance of the action. For `SfMovitAction`, `invoke()` is defined as:

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

In the file defining the action, we can add code to be executed when the file is loaded into Saphira, using then function `sfLoadInit`. Just as in making a C++ function available to Colbert via the `sfAddEvalFn` interface, we make an action available to Colbert using the `sfAddEvalAction` interface. Here is how it's done for `SfMovitAction`:

```
SFEXPORT void                 // define interface to Colbert here
sfLoadInit ()
{
  sfAddEvalAction("Movit", (void *)SfMovitAction::invoke, 2, sfINT, sfINT);
}
```

Just as for functions, actions available in Colbert take argument that are typed. Here, there are 2 such arguments. There is no need to give a return type for the `invoke()` function, since it is assumed to be a pointer to an action object.

4.6 Invoking the Action

Once the action is defined and added to Saphira and Colbert, it can be invoked from Colbert using the `start` command, just like a Colbert activity.

The complete form of start is shown in the Colbert User's Manual. To start up the action, just use

```
start Movit(1000,95);
```

which will start up the action with a distance argument of 1000 and a heading argument of 95. An example of invoking the Movit action from Colbert is given in the tutor/movit/init.act file, as the activity movit_act:

```
act movit_act(int dist)
{
  start Movit(dist, 90) priority 10;
  remove Movit;               // get rid of the action
```

```
  stop;                          // stop the robot
}
```

Here the activity invokes Movit, then waits until it is completed before removing it and halting the robot. The appearance of `deactivate()` in the body of `SfMovitAction::run()` comes into play here: the Colbert activity will stay at the `Movit` action until it deactivates.

From C++ programs, the action can be invoked by using the corresponding `sfStartTask()` function. For example, here is a typical invocation:

```
SfStartTask("Movit", NULL, 0, 10, 0, 1000, 95);
```

Here the second argument is the instance name; the default NULL means it will be the same as the schema name. Timeout, priority, and suspension arguments are required, followed by the action parameters. For a complete description of `sfStartTask`, see the Colbert User's Manual.

4.7 Turning Behavioral Actions On

Behavioral actions and direct actions can conflict if they are invoked at the same time. For example, suppose we define a bump-and-go activity to turn the robot away from obstacles that it bumps into. This activity uses direct actions to back up and turn. But the robot may be under the control of behavioral actions when it encounters and obstacle: it may be heading towards a goal position, using something like the SfMovitAction. Now the direct actions and the behavioral actions are both trying to control the robot. What happens?

In situations like this, the direct actions always take priority. While the robot is executing direct actions, it turns off the output of behavioral actions. In fact, whenever a direct action is executed, behavioral actions remain off until they are explicitly turned back on.

To turn behavioral actions on, use the `behaviors` command in Colbert, or the longer C++ function:

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