RoboCup 3D Simulation HowTo

Little Green BATS

15th January 2008

Contents

Introduction

Robocup 3D Simulation

RoboCup is the largest AI/robotics related event in the world. It was started in 1996 with a highly ambitious goal: to propel the fields of Artificial Intelligence and robotics such that in 2050 a team of robotic players could defeat the human soccer world champions. RoboCup consists of several 'leagues' where different types of robots focus on different aspects of mechanics and/or cognition needed to achieve this goal.

One of these leagues is the Simulation League. Like the name suggests, there are no real robots in this league. An artificial soccer game, including the field, the ball and the players, are simulated on one or several computers and the results are visualized on a screen/beamer. The main focus of this league is to research higher cognitive abilities, like planning and team play, but also offers a low cost entry to the world of RoboCup for teams with a small budget.

At first, the simulation was restricted to 2 dimensions. However, in 2003 a begin was made with a new simulator that included the third dimension height and the RoboCup Simulation League's 3D challenge was born. The first few years the agents in the simulation were 3D versions of their 2D counterparts: spheres instead of circles, but in 2007 a giant leap forward was made by introducing complex humanoid agents.

Little Green BATS

The Little Green BATS is the first and so far only Dutch team in the 3D simulation league. We are a group of graduate students from the department of AI at the University of Groningen, The Netherlands. Our team name is derived from the fact that the first 3D agents in the league were balls and from the very philosophical observation that Balls Are Truly Spheres (BATS). This abbreviation reminded us of our favorite song 'Little Green Bag' by The George Baker Selection and so the whole team name was born.

We entered the competition for the first time at the 10th edition of RoboCup at Bremen, Germany. unfortunately our hard work didn't pay off that time: already in the second round we got eliminated. However, after this we had a good base to build upon and the good time we had at the event and the nice community inspired us to continue and work hard for another year. This turned out to be defiantly worth it, because in 2007 in Atlanta we managed to become vice world champions!

This How-to

This document describes the workings of the 3D simulation and of the architecture created and released by the Little Green BATS. It has the intention and hope to get new or existing teams started in the league.

The rest of this document is structured as follows. Chapter 1 describes the installation process of the 3D simulation server and of the BATS agent architecture. In chapter 2 the operation of the simulation server is explained. The theoretical base of the BATS agent architecture is handled in chapter 3. Finally, a step by step tutorial on the different parts of the BATS agent architecture is given in chapter 4.

Chapter 1

Installation

1.1 SimSpark Simulation Server

This explains how to install the Robocup 3D simspark simulation server on Ubuntu (so not the old spheres server using spades). This should work on Dapper Drake (6.06), Edgy Eft (6.10) and Feisty Fawn (7.04). The steps should be similar on other distributions like Fedora, Suse and especially Debian, so hopefully it's also useful for other people.

1. Enable Universe and Multiverse repositories:

   ```
   $ sudo gedit /etc/apt/sources.list
   [ Follow instructions in the file to enable the Universe
   and Multiverse repositories, save and exit ]
   $ sudo apt-get update
   ```

2. Install dependencies:

   ```
   $ sudo apt-get install g++ ruby1.9 ruby1.9-dev libode0-dev
   libboost-dev libsdl-dev libfreetype6-dev libdevil-dev
   autoconf automake1.9 libtool freeglut3-dev tetex-extra cvs
   xlibs-dev libtiff4-dev libslang1-dev
   $ sudo rm /usr/bin/ruby
   $ sudo ln -s /usr/bin/ruby1.9 /usr/bin/ruby
   $ sudo ln -s /usr/lib/libruby1.9.so /usr/lib/libruby.so
   ```

3. Check out the source from the source forge CVS repository:

   ```
   $ cvs -d:pserver:anonymous@sserver.cvs.sourceforge.net:
   /cvsroot/sserver login
   [ when asked for a password, just press enter]
   $ cvs -z3 -d:pserver:anonymous@sserver.cvs.sourceforge.net:
   /cvsroot/sserver co -P rcsoccersim/rcssserver3D
   ```

4. Build and install the soccer server: (optionally add –enable-debug=no and new versions don't need –enable-kerosin)

```
$ cd rcsoccersim/rcssserver3D/
$ ./bootstrap
$ ./configure
$ make
$ sudo make install
```

5. Make sure the linker can find your shared libraries:

```
$ sudo gedit /etc/ld.so.conf
[add the line '/usr/local/lib' if it isn't already there,
save and close]
$ sudo ldconfig
```

6. Run the simulation:

```
$ simspark
```

1.2 BATS Agent Architecture

1. Install dependencies:

```
$ sudo apt-get install libxml2-dev
```

2. Check out the source from the source forge CVS repository:

```
$ svn co https://littlegreenbats.svn.sourceforge.net/svnroot/littlegreenbats l
```

3. Build and install the BATS agent:

```
$ cd littlegreenbats/trunk/
$ ./configure
$ make
```

4. Run an agent:

```
$ ./humanoidbat -c lgb.xml -u 2
```

Chapter 2

SimSpark Simulation Server

The simulation is generated by the SimSpark Soccer Simulation Server. Agents can enter the simulation by communicating with the soccer server using tcp sockets and a hierarchical parentheses based communication protocol (think of LISP). The simulation maintains a physically realistic (although at the moment noiseless) world, existing of a soccer field and soccer(ro)bots. One agent program represents one soccer robot in the simulation.

As of the World Championship of 2007 the simulation can only reliably run two agents per team and still had some hiccups in the form of physics errors causing exploding body part. But we are confident that these will be resolved in the near future.

2.1 The Soccer Field

Currently (RoboCup 2007) the soccer field is a rectangular field of 50 meters by 32 meters length and width. In each corner there resides a flag and on each side there are two goal posts. The size of the field is subjected to change and probably won't stay the same for very long. Luckily the server notifies the agent of the size of the field at the initiation of communication.

2.2 The Soccerbot

The soccerbot is a humanoid robot, loosely based on Fujitsu's HOAP-2 robot.

Chapter 3

BATS Agent Architecture

After setting up a large, behavior driven system for the 2006 RoboCup competition in Bremen, we decided that although dividing the code up into behaviors was good, the architecture was missing some crucial thing. The 2006 robots had a tendency to switch behaviors to quickly, and would also try to combine tasks that shouldn't be combined (like attacking and defending at the same time). With this in mind, we created a list of minimal requirements for our architecture:

- An agent should be able to commit to a task, but never get completely stuck in it. For example: *Keep getting up until you are done.*
- An agent should be able to decide which behavior best fits the situation. For example: *Dont' walk to the ball, when the play mode doesn't allow you to.*
- Behaviors should be groupable, to keep certain behaviors from interfering with others. For example: *Don't try to stand up, when you want to lie down on the ground.*
- Some behaviors are only usefull after running other behaviors. For example: *Don't try to go through a locked door, before unlocking it.*

The Little Green BATS architecture allows the agent programmer to group behaviors in a flexible hierarchy, commit agents to behaviors, and decide which of a group of applicable behaviors is the best. To support all of these features, the architecture introduces a hierarchy of behaviors. Each behavior can have sub-behaviors in steps and slots. Steps are used to create sequences of behaviors: when the next step is possible, it is run. Slots are contained in steps and allow for behaviors to compete (share a slot) or run in parallel (each behavior has it's own slot, and these slots share a step).

Because the architecture has no knowledge of the applicability of a behavior, the behavior will have to export this information in a uniform interface. This is known as the capability. The capability of a behavior, tells the architecture how likely (and how certain) a behavior thinks it can achieve it's goals when executed.

The architecture can choose between competing behaviors by sorting the capability of the competing behaviors and choosing the first in the list. This sorting method is something which is implementation specific .

Chapter 4

Tutorial

The BATS agent architecture consists of several parts and layers. This tutorial guides you through using these parts step by step. All lower layers are independent of the higher layers. So if you don't need all, you can skip the later sections. If you are only interested in an easy interface with the simulation server, reading section 4.1 will suffice. If you want to start quickly with a working agent, you can skip until section 4.4 for now. However, the agent template described there is based on the elements described in the sections before that, so be sure to read those too at some time, to fully understand how your agent works.

4.1 SocketComm

The lowest layer in our architecture manages the communication with the simulation server. This communication is done through TCP sockets and consists of S-expressions (predicates) that the agent and server send back and forth. The SocketComm handles the connection to the server and sending, receiving and parsing of the messages.

When creating a SocketComm you have to supply a hostname and a port number to connect to. When running the server on the same computer as your agent with default settings, these are 'localhost' and '3100'. After creating a SocketComm, the first thing to do is open an actual connection by calling connect:

```
SocketComm comm("localhost", 3100);
comm.connect();
```

The SocketComm keeps two internal message queues, one for input and one for output. These queues are filled and emptied, respectively, when calling SocketComm's update method. This call blocks until new data is received from the server:

```
comm.update();
```

SocketComm supplies several methods to place messages that should be sent to the server into the output queue. First of all, you can build your own predicate using the Predicate and/or Parser classes [1] and put it directly into the queue by calling the send method:

```
rf<Predicate> myPredicate = makeMyPredicate();
comm.send(myPredicate);
```

However, you can also leave the trouble of building the predicates to SocketComm by using the make*Message methods. These methods are also used by the init, beam and move* methods, which also place the messages directly into the queue for you.

The input queue holds the messages received from the server. To check whether there is a new message you can call hasNextMessage, to extract the next message you can use nextMessage:

```
while (comm.hasNextMessage())
  rf<Predicate> message = comm.nextMessage();
```

To conclude this section we present a typical way to have successful communication with the server:

---

[1] This method is not described in this HowTo. Look at the documentation in the source code for more info

```
// Create the SocketComm and connect
SocketComm comm("localhost", 3100);
comm.connect();

// Wait for the first message from the server
comm.update();

// Identify yourself to the server
comm.init(0, "MyTeam");

// Main loop
while (true)
{
  comm.update();
  while (comm.hasNextMessage())
    handleMessage(comm.nextMessage());
}
```

4.2 WorldModel

The SocketComm parses the S-expressions that the agent receives from the server into our Predicate structure. However, to extract useful data and keep a model of the world you still have to dig through these messages. In our architecture, WorldModel does exactly this, helping you avoid handling S-expressions/predicates completely on the input side. The WorldModel is a singleton object that encapsulates the SocketComm and provides a large number of methods to determine the world state.

Before the WorldModel can be used, it has to be initialized by supplying a SocketComm that is connected to the server. To use the WorldModel, you have to get a reference to the singleton object:

```
WorldModel::initialize(comm);
WorldModel& wm = WorldModel::getInstance();
```

The `WorldModel` should be updated at the beginning of every time step. This updates the `SocketComm` (so you don't have to call `SocketComm::update()` yourself anymore), reads all messages from them and updates the internal model according to them:

```
wm.update();
```

After this you can request the states of the game, the world and the agent itself by calling WorldModel's methods. For a description of these, please look at the documentation of the source code.

4.3 Cerebellum

The same way WorldModel keeps you from having to work with predicates on the input site, the Cerebellum overlays the output interface. It supplies more useful structures to define actions and the possibility to integrate actions from different sources in your agent. The Cerebellum has the Action substructure and a few of its derivatives with which you can make new actions. At the moment there are 4 different usable action types:

- `MoveJointAction` - Move a hinge joint or one axis of a universal joint
- `MoveHingeJointAction` - Move a hinge joint
- `MoveUniversalJointAction` - Move both axes of a universal joint
- `BeamAction` - Beam to a certain position

The Cerebellum is, also like the WorldModel, a singleton that can be retrieved by calling `Cerebellum::getInstance()` after which you can add actions to the Cerebellum. After all the sub parts have added their actions, you can call `outputCommands()` to send the collections through a SocketComm:

```
Cerebellum& cer = Cerebellum::getInstance();

rf<MoveJointAction> action =
  new MoveJointAction(Types::LLEG1, 0.1);
cer.addAction(action);

cer.outputCommands(comm);
```

When more than 1 action is supplied for a hinge joint or part of a universal joint, the given speeds will be averaged before sending to the server.

4.4 HumanoidAgent

As said above, this section will show how to quickly set up an agent that connects to the server, reads and parses messages that come from it and has a think cycle in which you can easily send actions back. This is done by using the `HumanoidAgent` class. It combines the classes described in the previous sections and creates an easy interface for new agents. To use it, you should create a class that inherits from it:

```
class MyAgent : public bats::HumanoidAgent
{
  public:
    MyAgent(std::string teamName)
    : bats::HumanoidAgent(teamName)
    {}
};
```

As you see, the constructor of the `HumanoidAgent` class expects a team name. You can also give it a host name and port to use to connect to and a uniform number. By default the latter is set to 0, meaning that the server picks a free uniform number. To run the agent, all you have to do is make a new object of your agent class and call the `run()` method on it:

```
int main()
{
  MyAgent agent("MyTeam");
  agent.run();
}
```

This runs the initialization code of the `HumanoidAgent`, which connects it to the server, sends the needed initialization message, and calls the `init()` method. Overload this method if your agent needs its own initialization code. After this, an infinite loop is started that updates the `WorldModel` and calls the `think()` method at each time step. If you want your agent to actually do something, overload this last method.

4.5 Behavior

The code in the previous section implements an agent which connects to te server, gets placed in the field and then stands there until it falls over. Now it is time to let your agent behave. The BATS agent architecture supplies a hierarchical behavior model to help you with this by following these steps:

- Implement seperate behaviors
- Create a configuration file placing the behaviors in a hierarchical structure
- Create and run the behaviors

4.5.1 Implementing a behavior

The BATS agent architecture gives the `Behavior` class as the building block of your agent. The basic thing to know is that a behavior receives a goal from a higher level behavior, creates subgoals that it wants to achieve in order to achieve the main goal and passes these goals to its subbehaviors. The behavior defines a sequence of slots in which arbitrary sub behaviors can be placed. This section shows how to create a behavior, section 4.5.2 will show how to define its super and sub behaviors.

Inherit Behavior

The first thing to do when creating a new behavior is to create a class that inherits from the `Behavior` class. It defines a couple of pure virtual methods that each behavior should implement (more on these later):

```
class MyBehavior : public Behavior
{
  virtual rf<Goal> generateGoal(unsigned step,
                                unsigned slot);
  virtual rf<State> getCurrentState();
  virtual ConfidenceInterval getCapability(rf<State> s,
                                           rf<Goal> g);

public:
  MyBehavior(std::string const &id,
             std::string const &playerClass);
};
```

You can do this by hand, but you can also use the utility script `createbehavior.pl` to do this for you:

```
$ cd util/
$ ./createbehavior.pl
[Enter the behaviors name (e.g. MyBehavior)]
```

This creates a directory with your behavior's name in the Behavior directory and fills it with a header file with the necesary method declarations and code files with basic implementations for these methods.

Constructor (defining slots)

After making the base of your behavior you have to implement the constructor, where you define the structure of the behavior's sub behaviors by creating slots. The `createbehavior.pl` script makes a start by creating a tree that defines a sequence of 1 step with 1 slot:

```
// Define the root, which is always a sequence:
d_tree = new AST::Node(sequenceType);
// Add one step to the sequence:
d_tree->addChild(new AST::Node(andType));
// Add one slot to the first step:
d_tree->getChild(0)->addChild(new AST::Node(orType));
```

If you need more steps in the sequence, you can add new conjunction nodes (`andType`) to the root, if you want to run sub behaviors in parallel in a step, you can add new disjunction nodes (`orType`). The following gives an example of a behavior with 3 sequence steps and 2 parallel slots in the second step:

```
// Define the root, which is always a sequence:
d_tree = new AST::Node(sequenceType);

// Add first step to the sequence:
d_tree->addChild(new AST::Node(andType));
// Add one slot to the first step:
d_tree->getChild(0)->addChild(new AST::Node(orType));

// Add second step to the sequence:
d_tree->addChild(new AST::Node(andType));
// Add two slots to the second step:
d_tree->getChild(1)->addChild(new AST::Node(orType));
d_tree->getChild(1)->addChild(new AST::Node(orType));

// Add third step to the sequence:
d_tree->addChild(new AST::Node(andType));
// Add one slot to the third step:
d_tree->getChild(2)->addChild(new AST::Node(orType));
```

getCurrentState()

The first virtual method to implement is `getCurrentState`. Different behaviors rely on different information of the world and/or agent state. In the BATS agent architecture a behavior defines this state itself in a standardized tree-based state description. This standardization is useful when using a learning algorithm to train different behaviors that use different state information. These state descriptions can also be used to define goals as states that should be reached, as we will see in the next sections.

A basic state description is a conjunction of the state of several variables. To cater for incertability, the state of a variable is defined as a range of possible values:

$$(0 \leq \mathit{BallDist} < 5) \wedge (10 \leq \mathit{OpponentDist} < 15) \tag{4.1}$$

A behavior's state is defined in its `getCurrentState` method. The `createbehavior.pl` script sets up a conjunction for you to place variables into. The following code shows how to create the state in 4.1:

```
rf<State> state = new State();
rf<OrNode> dis = state->addDisjunct();
rf<AndNode> con = dis->addConjunct();

con->addVar("BallDist", 0, 5);
con->addVar("OpponentDist", 10, 15);
```

getCapability()

Behaviors that are placed in the same slot compete with eachother for execution. They receive the same goal and are selected based on their capability to achieve that goal. Also, before advancing to the next sequence step, a behavior checks if the sub behaviors in that step have enough capability to finish that step. A behavior's capability is requested by calling its `getCapability` method, passing it a goal and the state it created in `getCurrentState`. The behavior returns an estimate of its capability of achieving the goal, ranging from -1 to 1, with a confidence interval that depicts the accuracy of the estimate.

Commitment

As explained earlier a behavior can commit to its goal when it is chosen. By doing so, it lets the super behavior know that it will take some time to reach the goal and that in intermediate steps it could not be good to switch between behaviors. Commitment is requested by setting the d_committed member variable, which should be done in the behavior's overloaded `update` method:

```
void MyBehavior::update()
{
  Behavior::update();

  // Check if we are already committed
  // due to committed subbehavior(s)
  if (d_committed)
    return;

  // Commit if we can still reach our goal
  if (goalStillReachable())
    d_committed = true;
  else
    d_committed = false;
```

The `update` method is called at the beginning of each time step if the behavior could be selected to run that timestep. First of all the `Behavior::update` should be called. This updates the child behaviors, checks if they are committed and commits the behavior if it should commit if children are committed [2]. If this is the case you can choose to override this, but usually you just return.

Next you check whether the behavior should commit to its goal. Usually this is the case when the goal can still be reached. It is important to also reset the flag when a goal is no longer reachable, as shown in the example, to prevent a behavior to lock the agent in useless behavior. Note that the behavior's goal (`d_goal`) is the goal received in the previous timestep.

4.5.2 Configuration XML setup

The humanoidbats agent has to have an XML configuration file. This file is used to structure all the behaviors in the binary together. The XML file will define all the steps and slots and will define a root behavior which is at the top of the behavior hierarchy. The general XML layout contains:

- The root conf element
- The player id (unum) to class type coupling
- The player class defenitions
  - The behaviors element
    - Behavior elements
      - Behavior parameters
      - Behavior slots
- XIncludes

---

[2] Abbreviated to SCICC (Should Commit If Children Commit)

The root conf element

Every XML file has a root element, in the configuration XML files this is the conf element. If the file includes other XML files, make sure to include the XInclude namespace. The most common header will therefore contain:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<conf xmlns:xi="http://www.w3.org/2003/XInclude">
```

The player id (unum) to class type coupling

All the players on the field share the same XML configuration files as they are all instances of the same command. To allow for the different player types to share the same XML, the playerclass has been added. Every agent has a playerclass, and all player classes have their own behavior hierarchy. To define which player class is used for which unum, using the following code:

```
<player id="1" class="attacker" />
```

The player class definitions

The player class element holds the behaviors for the player class. Every player class has an id and contains the behaviors.

```
<player-class id="attacker">
```

The behaviors element

The behaviors element is a simple container element for all the behavior elements. It currently has no attributes, but must contain at least one behavior with the id win.

Behavior elements

The behavior elements contain all the information for the behavior structure. Every behavior has a type, that names the name of the behavior in the binary, and an unique id, which is used to reference it from other parts of the XML. The id only has to be unique within the behaviors parent element, not the whole configuration XML.

```
<behavior type="MyBehavior" id="mybehavior">
```

Behavior parameters

Behavior parameters are behavior specific and are used in by the binary behavior to make it more general in use. All parameters for a behavior are kept inside the *param* element. This element can contain any type of element which is totally up to the binary behavior. An example of is:

```
  <param>
        <deg>10</deg>
        <joint slot="0">&rleg5;</joint>
        <joint slot="1">&lleg5;</joint>
  </param>
```

Behavior slots

Every behavior can have other behaviors below it. These other behaviors can be run in parallel, in sequence, or in competition. The slots are created with a *slot* element which is a direct descendant of the behavior element. Each slot has an index attribute which describes where it is placed below the behavior. Each slot, in turn, contains any number of references to behaviors defined in the configuration.

To reference to a behavior, a *behavior* element should be used with a *refid* attribute containing the XML configuration id of the behavior.

A typical example:

```
<slot index="0-0">
  <behavior refid="mybehavior" />
</slot>
```

4.5.3 Creating and Running Behaviors

Now you have defined all your behaviors and set up the hierarchy in your XML configuration file. The last thing to do is to have your agent create the correct behavior tree based on the configuration file and run the behaviors every time step. The first you simply do by loading the configuation file with the `Conf` class and then call the static `Behavior::createBehaviors` method:

```
void MyAgent::init()
{
  Conf::initialize("myconf.xml");
  Behavior::createBehaviors();
}
```

As you can see, your agent's `init` method is a good place to do it. Running your behaviors consists of running the root behavior, which in its turn runs its subbehaviors, et cetera, until the lowest, primitive behaviors are run. Just select your root behavior (we like to call it the 'win' behavior and is the behavior with id 'win') and run it:

```
void MyBehavior::think()
{
  rf<Behavior> win = Behavior::getWin();
  win->update();
  win->setGoal(new Behavior::Goal);
  win->achieveGoal();
}
```

After the behavior tree is processed, the primitive behaviors that have actions to send to the server are collected in the list of Action Command Behaviors. You should request these, collect their actions and send them to the Cerbellum:

```
void MyBehavior::think()
{
  ...
  Cerebellum& cer = Cerebellum::getInstance();

  std::set<rf<Behavior> > acBehaviors = Behavior::getActionCommandBehaviors();
  for (set<rf<Behavior> >::iterator iter = acBehaviors.begin(); iter != acBehaviors.end(); ++
   cer.addAction((*iter)->getAction());

  cer.outputCommands(d_comm);
}
```

Appendix A

List of joints

&head1	Torso to head, X-Axis (not used)
&lleg1	Torso to left hip, Z-Axis (twist left/right)
&lleg2	Left hip to Left thigh, X-Axis (backward/forward)
&lleg3	Left hip to Left thigh, Y-Axis (spread/close)
&lleg4	Left thigh to Left shank, X-Axis (stretch/bend)
&lleg5	Left shank to Left foot, X-Axis (toes down/toes up)
&lleg6	Left shank to Left foot, Y-Axis (left/right)
&rleg1	Torso to right hip, Z-Axis (twist left/right)
&rleg2	Right hip to Right thigh, X-Axis (backward/forward)
&rleg3	Right hip to Right thigh, Y-Axis (spread/close)
&rleg4	Right thigh to Right shank, X-Axis (bend/stretch)
&rleg5	Right shank to Right foot, X-Axis (toes down/toes up)
&rleg6	Right shank to Right foot, Y-Axis (left/right)
&larm1	Torso to Left shoulder, X-Axis (forward/backward)
&larm2	Torso to Left shoulder, Y-Axis (out/in)
&larm3	Left shoulder to Left upper arm, Z-Axis (twist left/right)
&larm4	Left upper arm to Left lower arm, X-Axis
&rarm1	Torso to Right shoulder, X-Axis (forward/backward)
&rarm2	Torso to Right shoulder, Y-Axis (out/in)
&rarm3	Right shoulder to Right upper arm, Z-Axis (twist left/right)
&rarm4	Right upper arm to Right lower arm, X-Axis

Bibliography

[1] The Little Green BATS homepage, `http://www.littlegreenbats.nl/`.