libbats manual

Little Green BATS

2nd November 2008

Contents

Chapter 1

Introduction

This is the manual of `libbats`, the RoboCup 3D Simulation agent library, developed by the RoboCup 3D Simulation team the Little Green BATS. This library can freely be used, both as in free beer and free speech, to create your own (team of) RoboCup 3D simulation agent(s) and to do research and enter competitions with. It supplies some modules to get you started quickly, is generic enough to add any algorithm, behavior model or learning method and still gives you detailed control of the basic if required.

This manual is intended to give you an overview of the library and to get you familiar with the structure and use of its parts. You should have your own agent running within a few lines of code. More detailed information on all possibilities can be found in the documentation in the source code. If you want you can dig through all this code, but we suggest you install doxygen [1] instead. If you do this and follow the installation instructions in chapter 2, you will found this documentation nicely formatted with HTML in the `docs/html` directory.

The next chapter will show how you can get the library ready for use. Chapter 3 takes you through the steps of creating your first agent. The final chapter discusses the most important modules in more detail. If something is still unclear, you found a bug or just have a question related to this library, do not hesitate to contact us. Good luck and happy coding!

[1]http://www.stack.nl/ dimitri/doxygen/

Chapter 2

Installation

2.1 Compile and install from source

1. Unpack the tarball:

   ```
   $ tar xzvf libbats.tar.gz
   ```

2. Run the `configure` script. If run without parameters it will be installed in the system's default directory (usually `/usr/local`). If you want to use a different directory you can use the `--prefix` argument:

   ```
   $ ./configure --prefix=/foo/bar
   ```

3. Build the library:

   ```
   $ make
   ```

4. Install the library (make sure you have write privileges to the install directory):

   ```
   $ make install
   ```

Chapter 3

Quick Start

This chapter is intended to show how easy it is to create an agent using libbats. By following these steps you will recreate the simple Hello World example agent that is supplied with the library. For more detailed information on the used modules, or when you only need a small part of the library like communication with the simulation server, see the next chapters.

The base of a `libbats` agent is the `HumanoidAgent` class. This class initializes all parts of the library and supplies a simple life cycle for your agent. So let's start by creating your own agent class by extending `HumanoidAgent`. Of course we have to create a constructor and the `HumanoidAgent` class requires that your agent defines an `init()` and a `think()`. Listing 3.1 shows what your header file may look like. Now what should these methods do?

`MyAgent()` The constructor should give some initialization information to the constructor of the base class `HumanoidAgent`. At least the name of your team should be supplied, but you could also set parameters such as the host and port to connect to. See the following chapters and details in `HumanoidAgent`'s class documentation.

`init()` This method is called once after the agent is created, a connection to the simulator is made and all parts of the library are initialized. You can use this to initialize your own things, like a localization module, movement generators, et cetera.

`think()` Here is is where you put your agent's brain. After the agent is started and initialized, this method is called at every think cycle, 50 times per second. When the `think()` method is called, new sensor information from the server is read and integrated in the `AgentModel` and `WorldModel` (more on these later). In this method your agent should decide what to do and make sure actions for the current think cycle are sent to the server.

At the moment we don't have our own fancy modules yet, so the constructor is empty, as well as the `init()` method. We do want our agent to do something

Listing 3.1: `myagent.hh`

```
#ifndef _INC_MYAGENT_HH_
#define _INC_MYAGENT_HH_

#include <bats/HumanoidAgent/humanoidagent.hh>

/**
  * My first agent
  */
class MyAgent : public bats::HumanoidAgent
{
  /**
    * Initialize agent
    */
  virtual void init() {}

  /**
    * Think cycle
    */
  virtual void think();

public:

  /**
    * The Constructor
    */
  MyAgent()
    : HumanoidAgent(std::string("MyTeam"))
  {
  }

};

#endif
```

Listing 3.2: `myagent.cc`

```
#include "helloworldagent.hh"
#include <bats/WorldModel/worldmodel.hh>
#include <bats/AgentModel/agentmodel.hh>
#include <bats/Cerebellum/cerebellum.hh>

using namespace bats;

void HelloWorldAgent::think()
{
  WorldModel& wm = SWorldModel::getInstance();
  AgentModel& am = SAgentModel::getInstance();
  Cerebellum& cer = SCerebellum::getInstance();

  double t = wm.getTime();

  double angles[4];
  angles[0] = am.getJoint(Types::LARM1)->angle.getMu();
  angles[1] = am.getJoint(Types::LARM2)->angle.getMu();
  angles[2] = am.getJoint(Types::RARM1)->angle.getMu();
  angles[3] = am.getJoint(Types::RARM2)->angle.getMu();

  double targets[4];
  targets[0] = 0.5*M_PI;
  targets[1] = 0.25*M_PI*sin(t/2.0*2*M_PI)+0.25*M_PI;
  targets[2] = 0.5*M_PI;
  targets[3] = -0.25*M_PI*sin(t/2.0*2*M_PI)-0.25*M_PI;

  double velocities[4];
  for (unsigned i = 0; i < 4; ++i)
    velocities[i] = 0.1 * (targets[i] - angles[i]);

  cer.addAction(new Cerebellum::MoveJointAction(
    Types::LARM1, velocities[0]));
  cer.addAction(new Cerebellum::MoveJointAction(
    Types::LARM2, velocities[1]));
  cer.addAction(new Cerebellum::MoveJointAction(
    Types::RARM1, velocities[2]));
  cer.addAction(new Cerebellum::MoveJointAction(
    Types::RARM2, velocities[3]));

  cer.outputCommands(SAgentSocketComm::getInstance());
}
```

cool however, so we will fill the `think()` method as shown in listing 3.2 to make him wave his arms at us. Let's look at what it all does.

lines 1-6 First include the header files of the modules that are used. All these are in the `bats` namespace, so in line 7 we import this namespace so we don't have to type `bats::` all the time.

lines 10-12 Here references to the used modules are requested. Most modules are singletons, which means there is only one instance of each class. The `WorldModel` and `AgentModel` do what you probably already expect they do: they are models of the world and of the agent. The `Cerebellum` is named after the part of your brain that handles control and coordination of your movements and is used to actually do stuff, as you will see later on.

line 14 One of the great and much used features of the `WorldModel`: telling time!

lines 16-20 We want our agent to wave his arms, by moving his shoulder joints. To do this it is useful to know the current state of these joints. This sounds like a job for the `AgentModel` and as you can see it is. The `Types` class defines all sorts of handy types used by several modules.

lines 22-26 Next we define the angles we want to move the joints to. Here a sinusoidal pattern is used to create a smooth, friendly waving behavior.

lines 28-30 The agent is controlled by setting the angular velocities of its joints, so here they are calculated based on the current and target angles and a gain factor.

lines 32-39 As mentioned earlier, the `Cerebellum` is used to act. It is fed with actions, in this case joint movements, but it also controls the other actuators like speech and beaming.

line 41 When the `Cerebellum` has gathered all actions it is time to send them to the simulation server. A `SocketComm`, in the form of the specialized `AgentSocketComm`, is needed for this, which handles the actual complicated communication through sockets.

And that's it! Almost. The only thing left to do now is to create an actual executable program that runs our agent. This is done by defining the standard `main()` method, creating an instance of the agent class and tell it to run, as shown in listing 3.3. Now compile these files with your favorite building method (ours is `ccbuild`[1]), fire up `simspark`, run the binary and wave back at your new friend!

[1]http://ccbuild.sourceforge.net/

Listing 3.3: `main.cc`

```
#include "myagent.hh"

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

Chapter 4

Modules

4.1 Rf

When working with pointers and dynamic memory allocation, things could get messy. A memory leak is easily produced and premature deletion of data can cause crashes. The library uses a garbage collection system to handle these problems, in the form of the `rf<T>` template class. This class is a wrapper around a pointer, keeps track of the number of times the pointer is used and deletes the data when this number is zero. It is mostly interchangeable with regular pointers, a pointer can be assigned to an `rf`, after which it can be used as a regular pointer:

```
rf<MyClass> myRf = new MyClass();
myRf->doSomething();
MyClass& myReference = *myRf;
```

It is required that a class for which `rf<T>` is used extends the `RefAble` class. This is needed for the reference counting:

```
#include <bats/RefAble/refable.hh>
class MyClass : public bats::RefAble
{
  // Member definitions
};
```

4.2 Singleton

Many modules of the library are so called singletons, so we will give a short introduction about what this is and how they are used. The singleton pattern is a design pattern that makes sure that there is no more than one instance of a certain class. Special measures are taken to prevent you from copying this instance or creating new objects of the class. This pattern is used in the library for modules for which it makes no sense and could cause problems if there are multiple instances of. This is for instance the case for modules keeping track of states, such as the `WorldModel` and the `AgentModel`, and a module for maintaining the connection with the server.

In this library, singletons are implemented by the `Singleton<T>` template class. It offers the static `getInstance()` method to request a reference to the single instance of class `T`, for instance `WorldModel`:

```
WorldModel& wm = Singleton<WorldModel>::getInstance();
```

For each singleton class of the library, a `typedef` is set for the `Singleton<T>` instantiation of that class, formed by prefixing the class name with a capital S. So another way to write the previous example would be:

```
WorldModel& wm = SWorldModel::getInstance();
```

If you want to use the `Singleton<T>` template to create singletons of your own classes, follow these steps:

- Your class must have a default constructor.
- Make all constructors, including the copy constructor, of your class private.
- Make the assign operator, `operator=`, private.
- Define `Singleton<T>`, instantiated with your own class, as friend of your class.

If you do this, your class definition will look like this:

```
class MySingleton
{
  friend class bats::Singleton<MySingleton>;
private:
  // Default constructor
  MySingleton() {}
  // Copy constructor, not implemented
  MySingleton{MySingleton const& other);
  // Assignment operator, not implemented
  MySingleton& operator=(MySingleton const& other);
public:
  // Some public stuff
};
// libbats style singleton typedef
typedef bats::Singleton<MySingleton> SMySingleton;
```

4.3 SocketComm

The lowest layer in the library 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. To make sure you don't have to worry about what this stuff actually is and does, the library offers you the `SocketComm`. This class handles the connection to the server and sending, receiving and parsing of messages. The library has two types of `SocketComm`s, an `AgentSocketComm` and a `TrainerSocketComm`. The first is used for regular communication between the server and the agent and will be described here. The latter, which can be used to set up training scenarios, will be discussed later on.

Before you can use `AgentSocketComm`, you have to supply a host name 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 this is done, the first thing to do is open an actual connection by calling `connect()`:

```
AgentSocketComm& comm = SAgentSocketComm::getInstance();
comm.initSocket("localhost", 3100);
comm.connect();
```

Note that the `AgentSocketComm` is a singleton. The `AgentSocketComm` keeps two internal message queues, one for input and one for output. These queues are filled and emptied, respectively, when calling `AgentSocketComm`'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 `AgentSocketComm` by using its `make*Message()` methods. And if you want it totally easy, use the `init()`, `beam()` and `move*()` methods, which not only build the predicates but 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();
```

[1]This method is not described in this manual. Look at the documentation of the classes for more info

To conclude and summarize this section, a typical way to have successful communication with the server is presented here:

```
// Get the AgentSocketComm, initialize and connect
AgentSocketComm& comm = SAgentSocketComm::getInstance();
comm.initSocket("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());
}
```

4.4 Cochlea

The `AgentSocketComm` parses the S-expressions that the agent receives from the server into a `Predicate` structure. However, to extract useful data you still have to dig through these structures. The `Cochlea` offers a layer over the `AgentSocketComm` to extract information from the predicates received from the server and present it in an easily accessible way.

Before using the Cochlea you have to initialize some parameters. You have to let it know what the name of your team is, so it can tell team mates and opponents apart:

```
Cochlea& cochlea = SCochlea::getInstance();
cochlea.setTeamName("MyTeam");
```

Next you have to set up translations for joint angle sensors. The `Cochlea` uses internal names for these, that may not be the same as the names used in the messages sent by the server. These internal names are "head1", "head2", "larm1" to "larm4" for the left arm, "rarm1" to "rarm4" for the right and "lleg1" to "lleg6" and "rleg1" to "rleg6" for the legs. The Nao robot used by the server for instance uses names of the form "laj1" and "rlj3", so these have to be translated:

```
cochlea.setTranslation("laj1", "larm1");
cochlea.setTranslation("rlj3", "rleg3");
```

This way the Cochlea can handle different robot models.

Now you can start using the `Cochlea` by calling `update()` every time a new message is received by the `AgentSocketComm`. This will integrate the information of the message after which you can request data with the `getInfo()` method, or one of the methods for more complex data:

```
comm.update();
cochlea.update();
// Get the polar coordinates of the ball
Vector3D polarBallPos = cochlea.getInfo(
                          Cochlea::iVisionBall);
// Go through all team mates
for (player_iterator iter = cochlea.beginPlayers();
     iter != cochlea.endPlayers(); ++iter)
  doSomethingWithPlayerInfo(*iter);
```

4.5 WorldModel

The raw data offered by the `Cochlea` may not be directly usable and perhaps you want to have some information that is deduced from these facts. This is exactly what the `WorldModel` and `AgentModel` are for. We will present the first one here.

To start, the `WorldModel` also needs to know the name of your team for some of its capabilities:

```
WorldModel& wm = WorldModel::getInstance();
wm.setTeamName("MyTeam");
```

Next, the model has to be updated at every cycle. The `WorldModel` extracts data from the `Cochlea`, so make sure it is updated before updating the `WorldModel`:

```
comm.update();
cochlea.update();
wm.update();

double t = wm.getTime();
```

The `WorldModel` has a large collection of methods that give information about the state of the world. One of the main uses is to get information about objects in the field, like the ball and players. This data is supplied by the `getObject()` and `beginPlayers()` / `endPlayers()` and `beginOpponents()` / `endOpponents()`. The most important part of this information is the position of the objects. The `WorldModel` takes the polar coordinates supplied by the `Cochlea` and transform them to 3D Cartesian coordinates in a coordinate system where the positive x-axis points to to the right side of the agent, the positive y-axis points forward and the positive z-axis is orthogonal to the field plane.

For information on the other data supplied by the `WorldModel`, see the class documentation.

4.6 AgentModel

What the `WorldModel` does for the agent's surrounding environment, the `AgentModel` does for its own state. It keeps track of joint angles and data of other sensors, as well as some higher level data like the position of the Center Of Mass (COM). The `AgentModel` also needs some initialization before it can be used. You have to tell it the uniform number of the agent, after which you should call the `initBody()` method:

```
AgentModel& am = SAgentModel::getInstance();
am.setUnum(unum);
am.initBody();
```

This loads an XML configuration file that contains the names, sizes and weights of the agent's body parts. It also uses this data to set up the translations for the `Cochlea` for you, so you don't have to do this by hand. If default settings are used, the default configuration file `conf.xml` is loaded, which is installed with the library and which imports the `nao_mdl.xml` file for each agent to get the description of the Nao robot model. For more information on loading XML configuration files and the robot model descriptions, see the documentation of the `Conf` class.

After initialization, the `AgentModel` is also ready to be updated and used:

```
comm.update();
cochlea.update();
wm.update();
am.update();

Vector3D com = am.getCOM();
```

4.7 Cerebellum

The `Cochlea` takes the trouble of having to deal with raw predicates on the input side. On the output side the `Cerebellum` is there for you. It supplies more useful structures to define actions and the possibility to integrate actions from different sources in your agent. The `Cerebellum` defines the `Action` substructure and a few of its derivatives with which you can make new actions. At the moment there are 5 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

`SayAction` Shout something

If you don't understand the difference between hinge and universal joints, don't worry. Just use `MoveJointActions` and let the `Cerebellum` handle the rest.

The `Cerebellum` again is a singleton that can be retrieved by calling `SCerebellum::getInstance()` after which you can add actions. After all the sub parts have added their actions, you can call `outputCommands()` to send them through an `AgentSocketComm`:

```
Cerebellum& cer = SCerebellum::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 joint, the speeds will be added together before sending the actions to the server.

4.8 HumanoidAgent

Now you have learned about how to initialize, update and maintain the several models of the library, you will learn how to forget all that. The `HumanoidAgent` class does this all for you. It connects the `AgentSocketComm`, initializes the `WorldModel` and `AgentModel` and updates all modules in the correct order at every cycle. See chapter 3 to learn how to use this class to set up your own agent.