Generation of Python Interfaces for RoboCup SPL Robots

David Claridge

davidc@cse.unsw.edu.au

University of New South Wales

February 2011

Contents

1 Introduction

Software used for participation in the annual RoboCup SPL Competition[2] is often developed at a rapid place in small highly-dynamic teams in the few months leading up to the competition. As such is essential to make use of efficient software engineering techniques, including:

- Using a streamlined build system
- Having a rapid test-development cycle
- Taking a fail-fast approach to developing new ideas

Perhaps the most dynamic component of an SPL software system is the behaviour subsystem, which is responsible for choosing the robot's intention based on the various sensor inputs and world models that are available to it. This module determines both the high-level strategy of a team of soccer-playing robots, as well as the particulars of individual skills such as dribbling, passing or walking to a desired location.

This module is evolved as the underlying infrastructure is improved, as well as in reaction to the strategies and capabilities of opposition teams at the competition. It is therefore necessary to utilise a framework that allows behaviours to be very rapidly modified, or even completely rewritten, during the week of the RoboCup competition.

One approach that satisfies all these goals is to use a dynamic programming language that lends itself to short development time, whilst trading off some performance compared to compiled machine code[4]. In this report we present a method for integrating the Python programming language with a C++ robotic software system, that provides full access to the C++ system's internal state as well as the ability to reload behaviours at runtime, greatly reducing developer overheads.

2 Related Work

A number of SPL teams, including rUNSWift, have attempted to make use of dynamic languages for describing robot behaviour in the past:

In 2010 rUNSWift utilised the Python C API to hand-craft wrappers for each of their C++ data types, allowing Python behaviours to be dynamically reloaded at runtime[5]. Unfortunately this approach required a huge amount of developer effort to maintain, since it was very easy for Python wrappers and C++ types to become out of sync. It also suffered from issues of memory leaks, due to the complexity of tracking all Python object references throughout the wrapper code.

In 2008 rUNSWift attempted to integrate the Ruby programming language into their C++ system, however complications using the Ruby C API led to this approach being abandoned before the 2008 RoboCup competition[1].

Since 2004 the SPL German Team, and more recently B-Human, the reigning world-champions, have been using a behaviour-description language called XABSL[6]. XABSL allows hierarchical finite-state machines to be described using a simple syntax; rUNSWift's 2010 team experimented with XABSL during their development, but found the overheads of integrating it with an existing C++ system to be too great. Similar to the rUNSWift 2010 Python method, it required manual conversion of complex data types into a format the XABSL can understand. Whilst not implemented by the German Team, a runtime re-loading system utilising inotify, would be possible with the XABSL runtime.

3 Implementation

This section will describe the various tools and methodologies that were used to integrate a Python interpreter with support for run-time reloading into the rUNSWift robotic software system.

Before diving into the details of the Python-C++ interface, one must first understand the overall architecture of the system and how its various components interact.

3.1 rUNSWift Software Architecture

The core functionality that allows rUNSWift's team of Nao robots to play soccer is encapsulated in the **runswift** executable program, which is a stand-alone application, linked against standard Linux system libraries such as libc, and third-partly libraries such as Boost, Eigen and libpython2.6. The **runswift** executable communicates with the robot's hardware via a shared-memory interface to a **NaoQi** extension library called **libagent**, details about which can be found in the rUNSWift 2010 Team Report[5].

There are two primary threads in the **runswift** executable: 'motion' and 'perception'. Motion is a real-time thread, running at 100Hz, which is responsible for the movements and stabilisation of the robot. The perception thread is responsible for sensing, world-modelling and decision-making, it runs at a maximum of 30Hz (IO-bound by the camera device), but usually fewer due to the computationally intensive nature of vision processing. It is the final decision-making part of the perception pipeline, called 'behaviour', which is the target for our automatic reloading using Python.

As can be seen in Figure 1, the behaviour module receives inputs from vision about detected features, and the world-model's state estimates of various objects and agents. The behaviour-loading sub-module is invoked on startup and whenever an change to the Python code is detected by the inotify watcher. Finally the behaviour execution sub-module will use the information provided to it, to choose an action command which can be posted for processing by the motion thread.

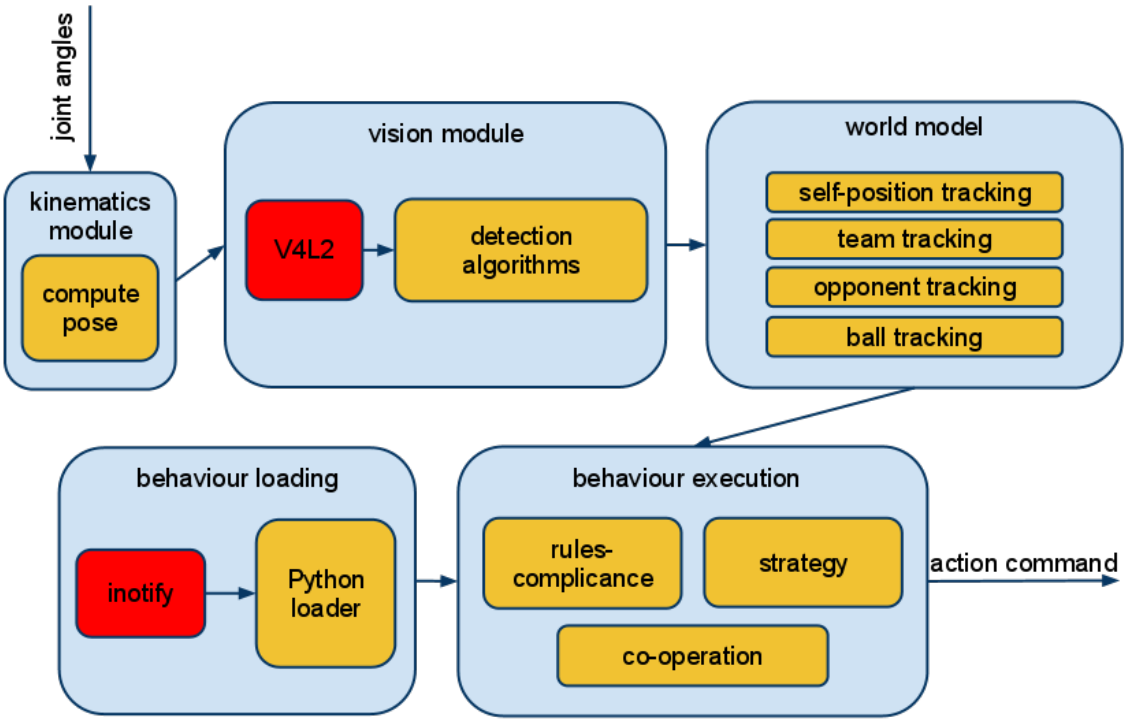

Figure 1: A data-flow diagram for the runswift perception thread

3.2 The *Robot* Python C Extension Module

In order for pure Python code, such as the Behaviour module (see Section 3.3) to access data stored in C++ data structures, a Python C Extension module must be used. The Python-C API provides methods that can generate `PyObject` structures from basic C types such as `ints`, `longs`, `floats` and null-terminated strings. To convert more complex types, one would usually create various Python proxy classes using the C API, and convert each of the fields to `PyObjects` on demand.

A major downside to this approach is the need to maintain this complex C extension module, and keep all the type wrappers synchronised with any changes to the underlying C++ data types. Furthermore, the C API requires the programmer to carefully track all references to objects, lest the Python garbage collector decide that a critical object is no longer needed and deallocates it, or conversely, if the reference count is not correctly decremented, a memory leak could easily be caused. With the Python Behaviour module running at approximately 30Hz, any memory leak in this part of the code will very rapidly cause a complete loss of all available system memory on the Nao, which only has 256MB of RAM.

To avoid having to maintain this unwieldy module, which in rUNSWift's 2010 code consisted of over 10,000 lines of code, whilst only providing wrappers for a small subset of data available to other C++ modules; we are now making use of the Simplified Wrapper and Interface Generate (SWIG), an open source toolsuite that generates bindings for various dynamic languages, including Python, given C/C++ header files and an interface descriptor file.

3.2.1 Wrapping the Blackboard with *SWIG*

The **runswift** executable utilises a central data store called the 'Blackboard' to share information between modules. Each module has an 'Adapter' which reads its required inputs from the Blackboard, calls the relevant module logic, and posts updated outputs from the module to the Blackboard. Providing Python behaviours with read access to the Blackboard is the simplest way to ensure that behaviour authors can access all the information they need.

In order to generate a Python C wrapper module, we provide SWIG with an interface descriptor file:

```
%module robot
%{
#include "blackboard/Blackboard.hpp"
%}
```

Listing 1: robot.i

The first directive specifies that SWIG will be generating a dynamic language module named 'robot'. This is followed by any C code that must be prepended to the generated module; in this case, we just need to include the header file for the Blackboard.

```
%include "std_string.i"
%include "std_vector.i"
%include "carrays.i"
%array_class(float, floatArray)
```

```
/* stdint */
%typedef signed char          int8_t;
%typedef short int            int16_t;
%typedef int                  int32_t;
%typedef long int             int64_t;
%typedef unsigned char        uint8_t;
%typedef unsigned short int   uint16_t;
%typedef unsigned int         uint32_t;
%typedef unsigned long int    uint64_t;

%include "eigen.i"
```

Listing 2: robot.i

By default, SWIG does not recursively process included header files, since this would lead to wrappers being generated for several very large standard C and C++ system libraries. In order for types not specified directly in Blackboard.hpp to be made available to Python code, we must provide SWIG directives to wrap them. Standard wrappers are available for C-style pointer arrays as well as C++ `std::string` and `std::vector` types. We also explicitly list typedefs for types used on the Blackboard that SWIG would not recognise.

We are using the libEigen matrix library's types in several of our data structures, however we do not want to wrap libEigen in its entirety, so we have written a SWIG interface descriptor file giving specific typemaps for the parts of libEigen that we use:

```
%typemap(out) Point {
   PyObject *x = PyInt_FromLong($1.x());
   PyObject *y = PyInt_FromLong($1.y());
   PyObject *p = PyTuple_Pack(2, x, y);
   Py_XDECREF(x);
   Py_XDECREF(y);
   $result = p;
}
```

Listing 3: eigen.i

SWIG's typemap system allows us to explicitly specify which Python C API calls should be used to convert an object of type `Point` (which is a typedef for an `Eigen::Vector2f`), into a `PyObject`. If we wished to return libEigen vectors from Python back to C++, would would also need to specify an 'in' typemap in addition to this 'out' typemap.

When SWIG encounters a type it is unaware of, it treats it as a raw pointer. This still allows Python code to pass references to objects from one C++ data structure or function to another, but not access their contents. Therefore, we include the header files for all the custom C++ types that are used on the blackboard, we that would like to have access to in Python, this will generate SWIG Python proxy classes for each of them:

```
%include "utils/body.hpp"
%include "utils/boostSerializationVariablesMap.hpp"
%include "utils/SPLDefs.hpp"
%include "utils/speech.hpp"
%include "perception/kinematics/Parameters.hpp"
%include "perception/vision/RobotRegion.hpp"
%include "perception/vision/WhichCamera.hpp"
%include "perception/kinematics/Pose.hpp"
%include "gamecontroller/RoboCupGameControlData.hpp"
%include "types/BehaviourRequest.hpp"
%include "types/Point.hpp"
%include "types/BBox.hpp"
%include "types/ActionCommand.hpp"
%include "types/ButtonPresses.hpp"
%include "types/Odometry.hpp"
%include "types/JointValues.hpp"
%include "types/SensorValues.hpp"
%include "types/RRCoord.hpp"
%include "types/AbsCoord.hpp"
%include "types/BroadcastData.hpp"
%include "types/RobotObstacle.hpp"
%include "types/FootInfo.hpp"
%include "types/BallInfo.hpp"
%include "types/PostInfo.hpp"
%include "types/RobotInfo.hpp"
%include "types/FieldEdgeInfo.hpp"
%include "types/FieldFeatureInfo.hpp"
```

Listing 4: robot.i

Finally, we instantiate any templates we are using on the Blackboard, and include the Blackboard header file itself:

```
namespace std {
   %template(FootInfoVector) vector<FootInfo>;
   %template(BallInfoVector) vector<BallInfo>;
   %template(PostInfoVector) vector<PostInfo>;
   %template(RobotInfoVector) vector<RobotInfo>;
   %template(FieldEdgeInfoVector) vector<FieldEdgeInfo>;
   %template(FieldFeatureInfoVector) vector<FieldFeatureInfo>;
   %template(AbsCoordVector) vector<AbsCoord>;
   %template(RRCoordVector) vector<RRCoord>;
}
```

```
%include "blackboard/Blackboard.hpp"
```

Listing 5: robot.i

This interface descriptor file can now be processed by swig2.0, with the following command in our CMakeLists.txt:

```
swig2.0 -Wextra -python -c++ -I${CMAKE_CURRENT_SOURCE_DIR} -o RobotModule.cpp $
    {CMAKE_CURRENT_SOURCE_DIR}/robot.i
```

Listing 6: SWIG command

This will generate two files: RobotModule.cpp, which contains the SWIG proxy classes using the Python C API, and robot.py, which is a pure-Python proxy to the C extension module. In a future version of SWIG it may be possible to import the C extension module directly, bypassing the need for a proxy module.

3.2.2 Modifications to Generated Code

The code generated by SWIG can be used as-is, with the exception of the Blackboard proxy class constructor. Usually SWIG is used to allow Python modules to create instances of C++ classes, but since we are embedding Python, we need to add a constructor parameter that allows us to pass a pointer to a C++ Blackboard object into the Python wrapper module. This can be achieved by applying the following patch to robot.py:

```
--- robot.py.before   2011-02-02 12:17:41.484491364 +1100
+++ robot.py    2011-02-02 12:17:58.585838992 +1100
@@ -1655,8 +1655,9 @@
     __swig_getmethods__ = {}
     __getattr__ = lambda self, name: _swig_getattr(self, Blackboard, name)
     __repr__ = _swig_repr
-    def __init__(self, *args):
-        this = _robot.new_Blackboard(*args)
+    def __init__(self, this=None, *args):
+        if this == None:
+            this = _robot.new_Blackboard(*args)
         try: self.this.append(this)
         except: self.this = this
     __swig_destroy__ = _robot.delete_Blackboard
```

Listing 7: robot.py.patch

How this constructor is utilised to create a SWIG wrapper of the existing Blackboard object will be demonstrated in Section 3.5.

3.3 The *Behaviour* Pure Python Module

The behaviour code itself resides in a series of Python files that make up the 'behaviour' module. These files are not pre-compiled, and are stored in the robot's home directory at runtime.

The C extension module described in Section 3.2 has wrapped two key C++ data types in Python proxy classes: Blackboard, and BehaviourRequest. The Blackboard is a central data store that behaviour will use to read the robot's state, in particular the outputs of the vision and localisation modules are of interest, so behaviour Python code will take a reference to the Blackboard as a parameter.

In order to give the motion module actuation commands, our Python behaviours return an BehaviourRequest structure, which contains the desired state of the walk, the head, the robot's LED indicators, and which camera to switch on.

The top level class in the Python Behaviour hierarchy is as follows:

```python
import robot
import sys

skillInstance = None

def tick(blackboard):
    skill = blackboard.behaviour.pythonClass
    global skillInstance
    if skillInstance == None:
        exec "from skills.%s import %s" % (skill, skill)
        skillInstance = eval(skill+"()")

    return skillInstance.tick(blackboard)
print 'Python Loaded'
robot.SAY('Python loaded')
```

Listing 8: behaviour.py

This top-level behaviour exists simply to delegate responsibility to lower level behaviours. The blackboard variable 'behaviour.pythonClass' is specified in a configuration file, and optionally overridden as a command line option, allowing developers to specify which behaviour they would like to run. The blackboard pointer is passed on to those lower-level skills, and they are also expected to return an BehaviourRequest structure.

A more substantial behaviour that may be called from this top-level behaviour and utilises several blackboard variables and action commands, is the go-to-ball skill:

```
import robot
import actioncommand
import math
from TrackBallSkill import TrackBallSkill

class GoToBallSkill(object):
   def __init__(self):
      self.trackBallSkill = TrackBallSkill()

   def tick(self, blackboard):
      behaviour = self.trackBallSkill.tick(blackboard)

      body = actioncommand.walk()
      if blackboard.localisation.ballLostCount < 10:
         rr = blackboard.localisation.ballPosRr
         x = rr.distance() * math.cos(rr.heading()) - 170
         y = rr.distance() * math.sin(rr.heading())
         if x > 250 or abs(rr.heading()) > math.radians(20):
            body = actioncommand.walk(int(x),0,rr.heading())
         else:
            body = actioncommand.walk(int(x),int(y),0)

      behaviour.actions.body = body
      return behaviour
```

Listing 9: behaviour.py

This behaviour demonstrates accessing the blackboard, delegating parts of the behaviour to lower-level skills (in this case the TrackBallSkill sets the head action command), and overriding a subset of the action command parameters, in this case only the ‘body’ action.

3.4 Directory Monitoring with *inotify*

In order to allow programmers to actively make changes to Python code whilst the robot is running, substantially reducing development time overheads, we use **inotify** to monitor the filesystem for changes. Inotify is a subsystem of the Linux kernel that provides a collection of system calls that user-space applications can use to be notified about filesystem events. In particular, we use `inotify_init()` and `inotify_add_watch()` to subscribe to chagnes that occur in the directory used for storing Python skills.

```
void PythonSkill::startInotify() {
    inotify_fd = inotify_init();
    int wd;
    wd  = inotify_add_watch(inotify_fd, path,
                                    IN_MODIFY | IN_ATTRIB | IN_MOVED_FROM | IN_MOVED_TO
                                        | IN_DELETE);
    if (wd < 0) {
        llog(ERROR) << "Failed to start watching direcotry: " << path << endl;
    }
    wd = inotify_add_watch(inotify_fd, (string(path) + "/skills").c_str(),
                                    IN_MODIFY | IN_ATTRIB | IN_MOVED_FROM | IN_MOVED_TO |
                                        IN_DELETE);
    if (wd < 0) {
        llog(ERROR) << "Failed to start watching direcotry: " << path << endl;
    }
    inotify_timeout.tv_sec = 0;
    inotify_timeout.tv_usec = 0;
}
```

Listing 10: PythonSkill.cpp

Each cycle, before the behaviour tick function is executed (see Section 3.5), the file descriptor provided by inotify is polled for information. If a new event is available, the affected filename is compared to a regular expression, which ensures it is indeed a Python script. If that is the case, the Python loading mechanism is called prior to that cycle's execution.

```
bool PythonSkill::inotify_Check() {
    bool reloadNeeded = false;
    FD_ZERO(&inotify_fdss);
    FD_SET(inotify_fd, &inotify_fdss);
    int selret = select(inotify_fd + 1, &inotify_fdss, NULL, NULL, &
        inotify_timeout);
    int i, len;
    if (selret < 0) {
        llog(ERROR) << "select on inotify fd failed";
    } else if (selret && FD_ISSET(inotify_fd, &inotify_fdss)) {
        /* inotify event(s) available! */
        i = 0;
        len = read(inotify_fd, inotify_buf, INBUF_LEN);
        if (len < 0) {
            llog(ERROR) << "read on inotify fd failed" << endl;
        } else if (len) {
```

```
            while (i < len) {
                struct inotify_event *event;
                event = (struct inotify_event *) &inotify_buf[i];
                if (event->len) {
                    boost::regex matchRegex(".*\\.py$");
                    if (regex_match(event->name, matchRegex)) {
                        llog(INFO) << "Detected change in " << event->name << endl;
                        reloadNeeded = true;
                        break;
                    }
                }
                i += sizeof(struct inotify_event) + event->len;
            }
        }
    }
    return reloadNeeded;
}
```

Listing 11: PythonSkill.cpp

With this functionality in place, a developer would typically make modifications to Python behaviours, and upload them to the robot while it is still running. Since the Motion module runs in a separate thread to Perception, the robot's walk engine is not interrupted and the loading of Python behaviours does not impact the Nao's stability.

3.5 Embedding, Loading, and Unloading

When the **runswift** executable starts, or when the Python code is changed on the robot at runtime (see Section 3.4), the Python C API is used to initialise the interpreter and load the necessary modules.

This functionality is encapsulated in a class called 'PythonSkill', which stores references to several key PyObjects, and SWIG run-time type information:

```
class PythonSkill : Adapter {
    ...
private:
    ...

    PyObject *behaviourModule;
    PyObject *behaviourTick;
    PyObject *pyKeyboardInterrupt;
    PyObject *pyBlackboard;

```

```
    swig_type_info *SWIGTYPE_p_Blackboard;
    swig_type_info *SWIGTYPE_p_BehaviourRequest;
};
```

Listing 12: PythonSkill.hpp

The behaviourTick object needs to store a reference to the top level method in the Behaviour module, which will be called each cycle. The pyBlackboard object will store a SWIG-wrapped proxy object to the blackboard, passed as the argument to behaviourTick.

First the Python interpreter needs to be shut down if it is already running, freeing any memory not yet claimed by the Python garbage collector, then the interpreter is re-initialised.

```
void PythonSkill::startPython() {
    if (Py_IsInitialized()) {
        Py_Finalize();
    }

    // Start interpreter
    Py_Initialize();
```

Listing 13: PythonSkill.cpp

When Behaviour code attempts to `import robot`, first the modules dictionary is checked, then pure Python modules and dynamically linked libraries are searched for on the `PYTHONPATH` and in `PYTHONHOME`. We directly initialise the C extension module defined in RobotModule.cpp, this loads the module into the Python modules dictionary so that it can be found at runtime.

```

    init_robot();
```

Listing 14: PythonSkill.cpp

For the behaviour module itself to be found, we need to add the directory on the robot where Python code is being stored to the `sys.path` object. This is done by borrowing a reference to the system path object, and calling the `append` method on it. We also obtain a reference to the `KeyboardInterrupt` exception, to be later used in error handling code.

With the path updated, the pure-python wrapper around the SWIG robot module can be imported and stored in the `robotModule` PyObject.

```
    PyObject *sysPath = PySys_GetObject((char *)"path"); // borrowed reference
    PyObject *appendRet = PyObject_CallMethod(sysPath, (char *)"append", (char *)"s", path);
    Py_XDECREF(appendRet);

    // Obtain KeyboardInterrupt exception
    PyObject *exceptionsModule = PyImport_ImportModule("exceptions");
    pyKeyboardInterrupt = PyObject_GetAttrString(exceptionsModule, "KeyboardInterrupt");
    Py_XDECREF(exceptionsModule);

    // Import robot
    PyObject *robotModule = PyImport_ImportModule(robotModuleName);
    PyErr_Check();
```

Listing 15: PythonSkill.cpp

With this module in hand, we obtain a reference to the Blackboard wrapper class, and call its modified constructor with a SWIG-wrapped Blackboard proxy as the first argument. To get this parameter, we use the SWIG_NewPointerObj method, with the blackboard pointer and the SWIG run-time type struct pertaining to the Blackboard class as arguments. We also load the run-time type information for a BehaviourRequest at this point, for use later when returning objects from Python behaviour code.

```

    // Obtain Blackboard class
    PyObject *robotBlackboardClass = PyObject_GetAttrString(robotModule, "Blackboard");
    PyErr_Check();

    // Find swig_type_info structures for our wrapped  classes
    SWIGTYPE_p_Blackboard = SWIG_TypeQueryModule(SWIG_Python_GetModule(), SWIG_Python_GetModule(), "Blackboard *");
    if (SWIGTYPE_p_Blackboard == NULL) {
       throw new std::runtime_error("Unable to find SWIG type for Blackboard");
    }
    SWIGTYPE_p_BehaviourRequest = SWIG_TypeQueryModule(SWIG_Python_GetModule(), SWIG_Python_GetModule(), "BehaviourRequest *");
    if (SWIGTYPE_p_Blackboard == NULL) {
       throw new std::runtime_error("Unable to find SWIG type for BehaviourRequest");
    }

    // Construct SWIG wrapper object around the real blackboard
    PyObject *pyBlackboardPtr = SWIG_NewPointerObj(SWIG_as_voidptr(blackboard), SWIGTYPE_p_Blackboard, 0);
    if (pyBlackboardPtr == NULL) {
```

```
        throw new std::runtime_error("Unable to create SWIG pointer wrapper
            around Blackboard");
    }
    pyBlackboard = PyObject_CallFunctionObjArgs(robotBlackboardClass,
        pyBlackboardPtr, NULL);
    if (pyBlackboard == NULL) {
        throw new std::runtime_error("Unable to create SWIG proxy around
            Blackboard pointer");
    }

    Py_XDECREF(robotBlackboardClass);
    Py_XDECREF(pyBlackboardPtr);
```

Listing 16: PythonSkill.cpp

Finally, the pure-Python behaviour module itself is imported, and we store a reference to the tick() function, to be called each cycle.

```

    pythonError = false;

    // Import behaviour
    behaviourModule = PyImport_ImportModule(behaviourModuleName);
    if (PyErr_Check()) {
        Py_XDECREF(behaviourModule);
        behaviourModule = NULL;
        pythonError = true;
    }

    // Obtain tick() function
    if (!pythonError) {
        behaviourTick = PyObject_GetAttrString(behaviourModule, "tick");
        if (PyErr_Check()) {
            Py_XDECREF(behaviourTick);
            behaviourTick = NULL;
            pythonError = true;
        }
    }

    if (pythonError) {
        SAY("Python import error");
    }
}
```

Listing 17: PythonSkill.cpp

At each step, the Python interpreter's error state is checked, and we abort after logging a stack trace and vocalising a warning to the user, in the case that one of the imports has failed. Most often this occurs due to a syntax error in the imported Python code, which is printed to the terminal on the robot, allowing a developer to fix it and upload a new version of the Python behaviours before continuing.

4 Maintenance and Future Work

The most likely addition of features to this system will be the introduction of new data types on the Blackboard that must be wrapped using SWIG. In most cases, simply adding the relevant header file to the SWIG interface descriptor file will suffice, however in some cases the type may need to be modified before becoming SWIG-compatible. Refer the SWIG documentation[7] for specifics of how to mangle header files that the generator finds troublesome.

In more unusual cases, such as with the Eigen matrix library, attempting to completely wrap the header file with SWIG will not be possible, so custom typemaps will have to be written for the needed classes or structures. Refer to the Python C API Documentation[3] for details on converting low-level types into their PyObject representations.

5 Conclusion

This project has provided an effective means for rUNSWift developers to rapidly evolve the behavioural software on the Nao humanoid robot, by taking advantage of the features of the Python scripting language, and the Linux inotify subsystem. Compared to previous methods it does not require active maintenance of the Python-C wrappers and converts, since these are generated by SWIG, making the system more robust as well as further saving developer's time.

References

[1] Adam Brimo. COMP3902 Project Report - RoboCup. `http://www.cse.unsw.edu.au/~robocup/2008site/reports/AdamBrimo-RobocupReport.pdf`, 2008.

[2] RoboCup SPL Technical Committee. Robocup Standard Platform League. `http://www.tzi.de/spl/`, 2010.

[3] Python Foundation. Python 2.6 C API Documentation. `http://docs.python.org/release/2.6.6/c-api/index.html`, 2011.

[4] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer magazine*, March 1998.

[5] Adrian Ratter, Bernhard Hengst, Brad Hall, Brock White, Benjamin Vance, David Claridge, Hung Nguyen, Jayen Ashar, Stuart Robinson, and Yanjin Zhu. rUNSWift 2010 Team Report. `http://www.cse.unsw.edu.au/~robocup/2010site/reports/report2010.pdf`, 2010.

[6] Max Risler and Oskar von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in xabsl. In *AAMAS08:Workshop on Formal models and methods for multi-robot systems*, May 2008.

[7] SWIG Development Team. SWIG 2.0 Documentation. `http://www.swig.org/Doc2.0/SWIGDocumentation.html`, 2010.