# A Framework for Advanced Robot Programming in the RoboCup Domain

*— Using Plug-in System and Scripting Language —*

Hayato Kobayashi [a,1], Akira Ishino [b], and Ayumi Shinohara [c]
[a] *Graduate School of Information Science & Electrical Engineering, Kyushu University*
[b] *Office for Information of University Evaluation, Kyushu University*
[c] *Graduate School of Information Science, Tohoku University*

**Abstract.** RoboCup is a competition for autonomous robots playing soccer that makes contributions to various Intelligent Autonomous Systems. In RoboCup, frameworks to support robot programming are important because we have to resolve complex difficulties by software, especially in the four-legged robot league, where we can never resolve these difficulties by hardware since only fixed hardware is available.

This paper describes an extensible framework which is suitable for advanced robot programming in the RoboCup domain. Our framework integrates a plug-in system and the scripting language Lua, which we embed in the system.

In our framework, modules are freely replaced, without changing the bindings for the scripting language. Therefore, in our framework, even programming beginners can contribute to the development of huge, complex, robot programs without difficulty. Since many students who are not familiar with practical programming often join the team only a short time before the RoboCup competition, our framework is a good choice.

**Keywords.** Framework, Plug-in System, Embedding Scripting Language, RoboCup

## 1. Introduction

RoboCup is a competition for autonomous robots which play soccer. Many researchers in the fields of Artificial Intelligence and Robotics attempt to make various Intelligent Autonomous Systems for use in this competition. Moreover, RoboCup is also a very interesting and challenging research domain, because it has a noisy, incomplete, real-time, multi-agent environment.

RoboCup has five different leagues: simulation, small-size robot, middle-size robot, four-legged robot, and humanoid league. We have participated in the four-legged robot league since 2003, as team *Jolly Pochie* [1]. Our league has its own specific difficulties, such as a camera with a narrow field of vision, quadrupedal locomotion, and limited resources. Since the hardware is fixed in an AIBO, which is an entertainment robot designed by Sony, without any alterations, we have to resolve the difficulties only by soft-

[1] Correspondence to: Hayato Kobayashi, E-mail: h-koba@i.kyushu-u.ac.jp.

ware. From a software standpoint, a robot program for the four-legged robot league may be more complex than that for the other leagues.

In RoboCup, especially the four-legged robot league, researchers must create complex robot programs for playing soccer. This is often a burden for researchers whose research topics are not closely related. Therefore, as with usual software engineering, many methods for reducing this burden are proposed in RoboCup [2,3,4,5,6]. We define overall systems which use these methods, or which make it easier to create robot programs from various perspectives, as *frameworks*. Other analogous frameworks are discussed in the next section.

In this paper, we describe our framework developed for the RoboCup competition. Our framework integrates a plug-in system and the scripting language Lua [7], which we embed in the system. Thanks to the excellent mechanisms supplied by Luabind [8], it becomes quite easy to bind C++ modules in this system. Thus, our framework enables us to create many modules easily and quickly without the difficulties associated with the bindings. As a matter of course, scripts in our framework can always access the methods of the modules. Therefore, we can develop each low-level module separately and independently, while other teammates are writing and adjusting high-level strategic scripts. This is quite advantageous to the development for the RoboCup competition, because even a new member of the team, who is neither familiar with practical C++ programming nor understands the whole complicated system developed so far, can contribute to adjustments of the parameters, examination of another strategy, and so on. Through these experiences, she/he will understand the whole system gradually and be prepared to join the development of the core parts later.

The rest of this paper is organized as follows. In section 2, we discuss related work. In section 3, we illustrate the overview of our framework. In section 4, we describe the details of the plug-in system, and in section 5, we describe a way to embed the scripting language Lua. In section 6, we explain the specification of our robot scripts in our framework. Finally, section 7 presents our conclusions and future work.

## 2. Related Work

There have been several studies done on frameworks and elemental techniques. The work in [2,3,5,6] shows multiplatform systems, which are also easy to program separately by using modular architecture. The work in [6] also used the concept of a "plug-in". The work in [4,9] shows frameworks where programmers can easily describe high-level processes. The work in [10,11,12] shows techniques to embed scripting languages whereby programmers can intuitively describe high-level processes. The work in [13,14] shows frameworks where programmers can generally describe low-level processes with a scripting language.

With respect to these related works, our framework has a very flexible and extensible architecture. Our framework integrates two techniques, that of a plug-in system and embedding a scripting language, without any disadvantages. There have been no studies that have tried to use these two techniques. Using our framework, all programmers from beginners to experts can collaboratively work toward one goal, which is simply to create the soccer robot programs.

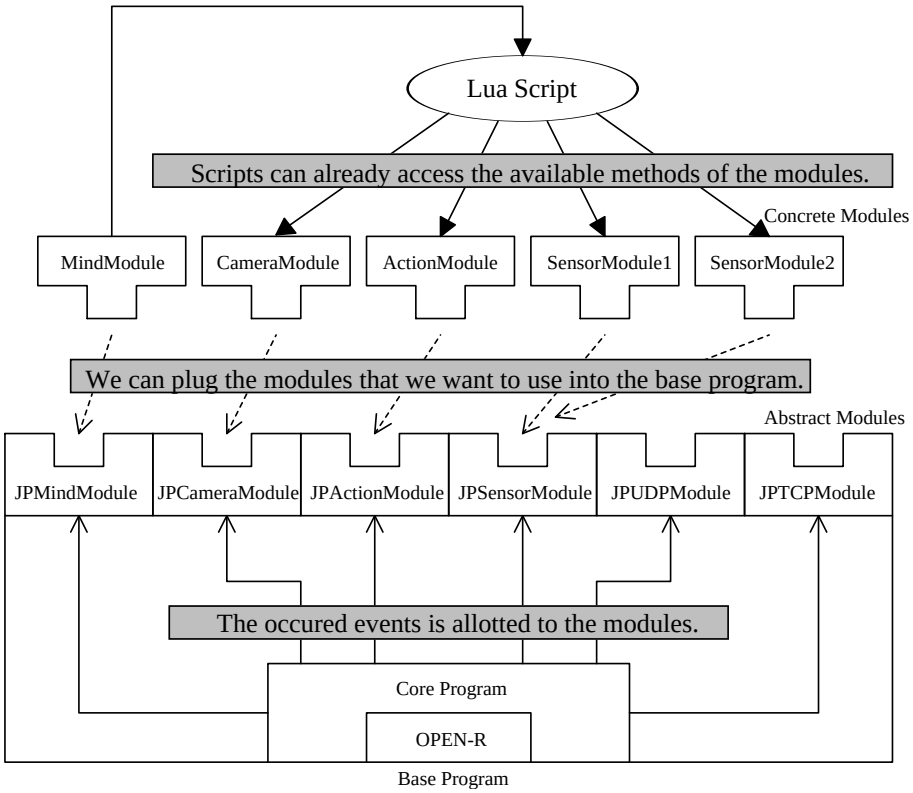


**Figure 1.** The overview of our framework.

## 3. Overview of Our Framework

Our framework is mainly based on two techniques. One is the plug-in system, and the other is embedding a scripting language. The plug-in system helps the development by allowing several programmers to collaborate. We can simplify each task by splitting a huge, complex process into individual functions. Embedding a scripting language helps us eliminate the trial and error process and the time it demanded. By making high-level scripts that do not need to recompile, our tasks can progress more efficiently.

Figure 1 shows some features of our framework. This is the programming procedure in our framework. We first separately create each low-level module. Then, we select some modules that we want to use, and plug the selected modules into the base program. For example, we would select a vision module, a localization module, a motion module, and so on, to make a robot that plays soccer. The robot program is generated easily by an automation tool for this procedure. We can build a binary program by compiling this robot program. Finally, we can write scripts for performing high-level processes by using the methods defined in the modules. In order to replace certain modules with others, once again, we only need to select modules, generate a program, and compile it. Nevertheless, we need not rewrite the script.

**Table 1.** The specifications of special functions.

| Module | Special Function | When is the function called? |
|---|---|---|
| `JPCameraModule` | `cameraNotify()` | Every 40 ms in sync with the CCD-camera |
| `JPMindModule` | `mindNotify()` | The same as `cameraNotify()` |
| `JPActionModule` | `actionNotify()` | When a set of joint angles are achieved |
| `JPSensorModule` | `sensorNotify()` | When sensor data is detected |
| `JPUDPModule` | `udpNotify()` | When UDP data is received |
| `JPTCPModule` | `tcpNotify()` | When TCP data is received |

## 4. Plug-in System

The concept of a "plug-in" has often been used in recent applications (*e.g.* in web browsers, drawing software, and integrated development environments). In RoboCup, Kleiner [6] introduced this concept. Our plug-in system is also similar to other existing plug-in systems. However, we must use OPEN-R SDK, which is the complicated programming interface that Sony is promoting for the creation of robot programs using AIBO. Therefore, we completely hide OPEN-R functions, data structures, and event handling in our framework. Consequently, we can avoid direct access to OPEN-R SDK in each module. Moreover, unit-test programs can be easily associated, even without access to OPEN-R SDK. Programmers can create modules with our user-friendly libraries, needing only an understanding of the simple rules used in our plug-in system.

As shown in Figure 1, our plug-in system consists of a base program and individual modules. The base program is the foundation of this system and common to all the robot programs. Concretely speaking, it is composed of classes wrapping the OPEN-R SDK. The most important class in the base program is the class `JPObject`, which inherits the class `OObject` in OPEN-R SDK. We create a robot program for AIBO by making a subclass of the class `OObject`, *i.e.* the class `JPObject` is the core of our robot programs. The class `JPObject` registers the instances of the constructed modules, and calls the modules every time certain events occur.

In order to create modules, we only need to make a subclass of *abstract classes* (*e.g.* `JPCameraModule` for processing camera images, `JPActionModule` for calculating joint angles during motion, `JPSensorModule` for managing information from sensors, and `JPMindModule` for developing strategies). The abstract classes have various *special functions* that are called when the class `JPObject` receives certain events. Table 1 shows the specifications of the special functions. For instance, the class `JPCameraModule` has the function `cameraNotify()` called every 40 ms in sync with the frame rate of the CCD-camera. That means, to get an image from the CCD-camera, we only need to make a subclass of the class `JPCameraModule`. In the same way, we can easily create various other modules. By selecting certain modules, we can make various robot programs not only for soccer players, but also for other events, such as the open challenge.

## 5. Embedding Lua

In the four-legged robot league, we only use AIBOs as our robots. Rebooting an AIBO requires a long time, and its battery may be drained in only 30 minutes. Therefore, some

teams [10,11,12] embedded a scripting language in their system. We also embedded a scripting language, Lua [7], in our system.

Lua is a scripting language designed to be embedded into C/C++. It has a small scripting engine with a simple and powerful syntax, and it can easily be embedded. The C/C++ program in which Lua was embedded can call Lua functions, read/write Lua variables, and register C/C++ functions.

As shown Figure 1, we embed Lua so that mind modules creating strategies can call Lua scripts and so that Lua scripts can call methods in other C++ modules.

In a mind module, the Lua function `mindNotify()` is called in the C++ member function `mindNotify()`. We can quite easily call a Lua function by using Luabind [8], which is a library that lets us intuitively create bindings between C++ and Lua. For example, in order to call the Lua function `mindNotify()` that has no argument and returns nothing, we only need to make the following call.

```
luabind::call_function<void>(Lua_Status, "mindNotify");
```

In order for Lua scripts to call methods in other C++ modules, it is necessary to bind the modules on the C++ side. This means we must register an instance of the modules, as well as information of the classes and member functions. Using Luabind, we can also easily bind C++ modules *in functions*. When embedding scripting languages (*e.g.* Lua, Python, and Perl), we typically must define global wrapper functions for functions that we want to bind. This means we rewrite the bindings whenever we exchange modules. However, Luabind can cut out this annoying task because it is implemented utilizing template meta programming. For instance, in order to register a class `ExampleModule` with public member functions `method1(const char* str)` and `method2(int x, int y)`, we only need to add the following code into any function.

```
module(Lua_Status) [
  class_<ExampleModule>("ExampleModule")
  .def("method1", &ExampleModule::method1)
  .def("method2", &ExampleModule::method2)
];
get_globals(Lua_Status)["exampleModule"] = this;
```

The last line registers the instance, assigning the `this` pointer of the class `ExampleModule` to the variable `"exampleModule"` on the Lua side. After the registration, we can call the functions within Lua scripts as follows.

```
exampleModule:method1("hello")
exampleModule:method2(10, 20)
```

Luabind can also register information regarding class inheritance. We need not bind in the class `AdvancedExampleModule` inheriting the class `ExampleModule`. Therefore, we need not rewrite our scripts even if we exchange these modules. The same is true for compatible modules in terms of bindings (*e.g.* `ExampleModule2` having the same methods that `ExampleModule` has).

**Table 2.** A part of the available C++ functions and variables in Lua scripts, which show that we can almost describe features for soccer players only by writing scripts.

| Instance | Available functions and variables |
|---|---|
| faceLED | setState(bitvector) |
| touchSensor | clickedBackFront(), clickedBackMiddle(), clickedBackRear(), clickedHead(), pressedBackFront(t), pressedBackMiddle(t), pressedBackRear(t), pressedHead(t), pressedChin() |
| soundPlayer | registWavFile(soundname, filename), playSoundOnce(soundname), changeVolume(volume), playSoundRepeat(soundname), playSoundStop() |
| visionBase | landmarksDetect(), ballDetect(), getLeftLineSlant(), getRightLineSlant(), getLeftLineIntercept(), getRightLineIntercept() |
| basicMotion | loadMotion(motionname, filename), stopAction(), playMotion(motionname, state), cancelAction(), playMotionLoop(motionname, state), swingHead(tilt1, pan, tilt2, nextstate), stopSwingHead(), cancelSwingHead(), loadGait(gaitname, gaitfile, odmeterfile), setGaitSource(gainame, rate), playGait(), playGaitLoop(), setGaitFirstFlag(flag) |
| detectBall | getL(), getPan(), getTilt(), getAdvisablePan(), getAdvisableTilt(), getInSightTilt1(), getInSightPan(), getInSightTilt2() |
| ballMCL | getX(), getY(), getPan(), getDistance(), isValid(), addXYV(x, y, dx, dy), shiftXYT(x, y, theta), update(), nupdate(n) |
| mcLocalization | getX(), getY(), getTheta(), update(), nupdate(n) |
| udpCom | udpSay(words) |
| psdSensor | getHeadPsdValue(), getHeadNearPsdValue(), getHeadFarPsdValue(), getBodyPsdValue() |
| gameControlData | firstHalf, kickOffTeam, secsRemaining, dropInTeam, dropInTime, myTeam.teamColour, myTeam.score, myTeam.player1.penalty, myTeam.player1.secsTillUnpenalised |
| accelSensor | getAccelX(), getAccelY(), getAccelZ(), getValueX(i), getValueY(i), getValueZ(i), getPosture() |

## 6. Specification of Our Robot Scripts

After embedding Lua, high-level processes can be described as scripts. In this accomplished framework we can create robot scripts with simple rules that everyone easily understand.

First, we have to understand the two functions `init()` and `mindNotify()` on the Lua side. The function `init()`, where we can initialize variables, is called only once at the beginning. The function `mindNotify()` is called by the member function `mindNotify()` in the mind module. That is to say, it is called every 40 ms.

Next, we have to know how to use the member functions that are bound in the C++ modules. As a concrete example, Table 2 shows a part of the available C++ functions

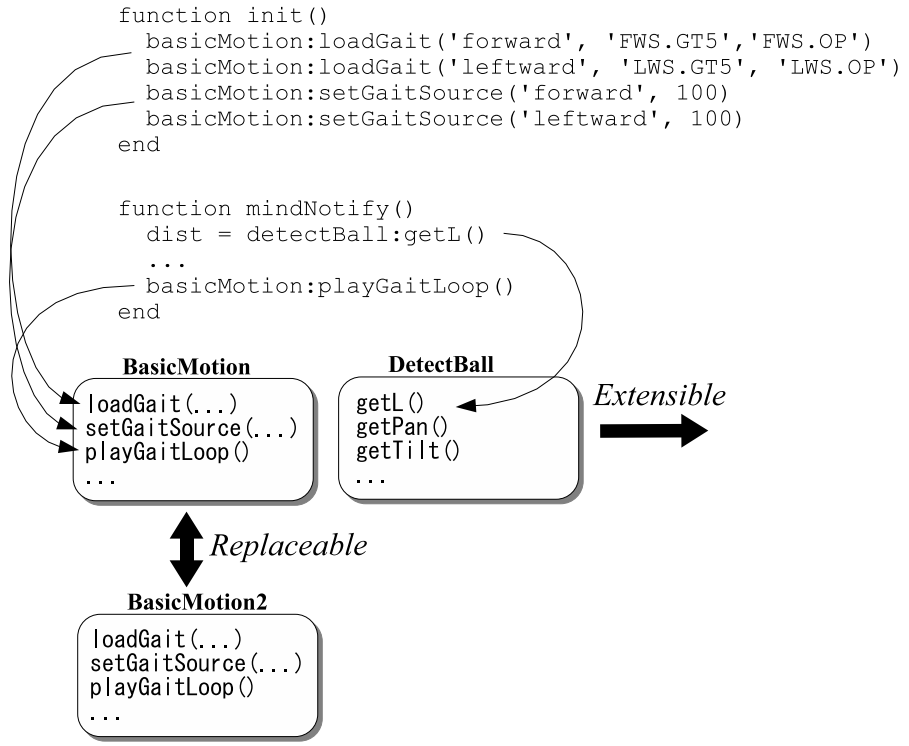


**Figure 2.** An example code of our robot script that makes a robot move diagonally-forward to the left and see a ball, where we use replaceable and extensible modules in our framework.

that we have developed so far. Using these functions, we can almost describe features for soccer players. That is to say, script programmers need not know complicated C++ programs at all. For example, Figure 2 shows a robot script that makes a robot move diagonally-forward to the left and see a ball. This figure also shows that the modules used there are replaceable and extensible.

## 7. Conclusions and Future Work

In this paper, we proposed an extensible framework which is suitable for advanced robot programming in the RoboCup domain. We could create many kinds of robot programs very easily, because the development process became dramatically more efficient. The plug-in system lets us divide a huge, complex robot program into many different modules that are manageable chunks that can be concentrated on. The embedded scripting language lets us efficiently create robot scripts. We could create various scripts without knowing the organization of C++ modules. We could also create various modules and immediately try them, because binding new modules was either simple or not necessary. Additionally, there was no need to rewrite scripts when replacing old modules with new modules.

We created approximately 130 modules and 350 scripts for the RoboCup 2005 competition in Osaka, Japan. Furthermore, we added many scripts and modules during the competition. It should be noted that many scripts were contributed by undergraduate students who had joined our team just two months before the competition, although all of them were unfamiliar with C++. The reason they were able to contribute is because they could write scripts intuitively. They were not concerned about complex C++ programs, and they could concentrate only on the team strategy.

Future work includes the development of the base program for robots other than AIBO. We will be able to reuse the same modules from AIBO. In the case of creating a humanoid robot program, we plan to reuse many modules, such as vision, while replacing the four-legged locomotion module with a two-legged locomotion module.

## References

[1] Jolly Pochie —Team for RoboCup Soccer 4-legged Robot League—. http://www.i.kyushu-u.ac.jp/JollyPochie/.

[2] Alessandro Farinelli, Giorgio Grisetti, and Luca Iocchi. SPQR-RDK: A Modular Framework for Programming Mobile Robots. In *RoboCup 2004: Robot Soccer World Cup VIII*, LNAI, pages 660–653. Springer, 2005.

[3] Thomas Röfer. An Architecture for a National RoboCup Team. In *RoboCup 2002: Robot Soccer World Cup VI*, LNAI, pages 417–425. Springer, 2003.

[4] Paul A. Buhler and José M. Vidal. Biter: a Platform for the Teaching and Research of Multiagent Systems' Design using RoboCup. In *RoboCup 2001: Robot Soccer World Cup V*, LNAI, pages 299–304. Springer, 2002.

[5] Holger Kenn, Stefano Carpin, Max Pfingsthorn, Benjamin Liebald, Ioan Hepes, Catalin Ciocov, and Andreas Birk. FAST-Robots: a rapid-prototyping framework for intelligent mobile robotics. In *Artificial Intelligence and Applications*, 2003.

[6] Alexander Kleiner and Thorsten Buchheim. A Plugin-Based Architecture for Simulation in the F2000 League. In *RoboCup 2003: Robot Soccer World Cup VII*, LNAI, pages 434–445. Springer, 2004.

[7] The Programming Language Lua. http://www.lua.org/.

[8] Luabind. http://luabind.sourceforge.net/.

[9] David S. Touretzky and Ethan J. Tira-Thompson. Tekkotsu: A Framework for AIBO Cognitive Robotics. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 1741–1743. AAAI Press, 2005.

[10] Gilad Buchman, David Cohen, Paul Vernaza, and Daniel D. Lee. The University of Pennsylvania RoboCup 2005 Legged Soccer Team. Technical report, Upenn, 2005.

[11] Ted Wong. The University of New South Wales School of Computer Science and Engineering. Technical report, rUNSWift, 2004.

[12] Manuela Veloso, Paul E. Rybski, Sonia Chernova, Colin McMillen, Juan Fasola, Felix von-Hundelshausen, Douglas Vail, Alex Trevor, Sabine Hauert, and Raquel Ros Espinoza. CMDash'05: Team Report. Technical report, CMDash, 2005.

[13] Jean-Christophe Baillie. URBI: Towards a Universal Robotic Low-Level Programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005.

[14] Douglas Blank, Deepak Kumar, Lisa Meeden, and Holly Yanco. Pyro: A Python-based Versatile Programming Environment for Teaching Robotics. *Journal on Educational Resources in Computing*, 3(4):1–15, 2003.