NaoTH Software Architecture for an Autonomous Agent

Heinrich Mellmann, Yuan Xu, Thomas Krause, and Florian Holzhauer

Institut für Informatik, LFG Künstliche Intelligenz, Humboldt-Universität zu Berlin,
Rudower Chaussee 25, 12489 Berlin, Germany.
{mellmann,xu,krause,holzhaue}@informatik.hu-berlin.de

Abstract. An appropriate architecture (i.e., framework) is the base of each successful heterogeneous software project. It enables a group of developers to work at the same project and to organize their solutions. From this point of view, the artificial intelligence and/or robotics related research projects are usually more complicated, since the actual result of the project is often not clear. In particular, a strong organization of the software is necessary if the project is involved in education.
Obviously, there is no perfect framework which could satisfy all the needs of the developers. In this paper we present a modular software architecture designed to implement an autonomous agent. In particular, it is used to develop software which is used simultaneously at several platforms (e.g., humanoid robot, simulated agent). One of the main aspects considered in our design is a strong code modularization which allows for re-usability, transparency and easily testing. Other important aspects are real-time applicability and simple usage.
This paper presents the main concepts and the particular implementation of the important parts. We also provide a qualitative comparison with other existing robotics frameworks.

1 Introduction

Since the Humboldt-Universität has a long history in RoboCup, there is a lot of experience and already existing code, especially from the GermanTeam[7] with its module based architecture. In the last years our research focused on developing a humanoid soccer agent running within the RoboCup *Simulation 3D* and *Standard Platform Leagues*. Thus, the presented framework evolved over the years based on the requirements and experience collected during this time. In order to play both leagues, the architecture strives to seamlessly integrate simulation system with real robotic hardware and allows simulated and real architectural components to function seamlessly at the same time. It also should be able to integrate existing available infrastructures.

In general, middleware is defined as *"a software which locates between OS and the application program, and indicates a library or a program etc. that offers the functions to improve convenience in a certain specific usage"*. Unfortunately, as stated by Hirukawa: *"Everyone agree that software should be modularized for*

Proceedings of SIMPAR 2010 Workshops
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Darmstadt (Germany) November 15-16, 2010
ISBN 978-3-00-032863-3
pp. 316-327

recycling and we should have a common architecture, problem is no one agree on how to do it". Thus, there are many middlewares have been implemented: OpenHRP (Open Architecture Humanoid Robotics Platform)[5], YARP (yet another robot platform)[6], Player[8], OROCOS (Open robot control software)[3], etc.. In [4] OpenRDK (Open Robot Development Kit) and a comparison of state of the art frameworks are presented.

NaoQi is a generic layer but it was created on Nao and fits to the robot. It allows homogeneous communication between different modules, homogeneous programming and homogeneous information shared with a blackboard. In OpenHRP there is unification of the controllers for both the simulation and the real counterpart, this leads to more efficient development of the controllers and the developed code is more reliable. The main features of YARP include support for inter-process communication, image processing as well as a class hierarchy to ease code reuse across different hardware platforms. Player defines a set of interfaces that capture the functionality of logically similar sensors and actuators, this specification is the central abstraction that enables Player-based controllers to run unchanged on a variety of real and simulated devices. OROCOS uses a CORBA-like component architecture with a hard real-time core. All these available frameworks share some common concepts including:

– modularity
– central data base (blackboard)
– communication/streaming of the data

The kind of the middleware is strongly influenced by the frame conditions of the project were it should be used. In our case the project is developed at an university and is mainly driven by the students. Thus, the main aspects defining the design of our software architecture are:

– as a research platform, the target of the project is not precisely defined
– different (concurrent) implementations for the same problem
– changing team members with different level of education
– volunteer team members (students)
– different operation systems
– cooperation between many researchers (which may be spatially separated)
– limited computational resources of the robot
– real time requirements

Based on this we can define some basic requirements for the architecture:

– modularity:
  the particular solutions for different (or the same) problems shouldn't affect each other and be easily exchangeable;
– as fast (small, simple) as possible:
  the program should run in real-time on the robot and the framework should be simple enough to be maintained by future generation of students;
– easy to use and easy to test:
  students with basic programming knowledge should be able to implement and test their algorithms;

– multi-platforms:
  the resulting program should run on different platforms (e.g., simulation, Nao, etc.) and on different operation systems (Windows, Linux, etc.);
– transparency:
  it should be possible to inspect the state of the program at any time during the runtime (e.g., which data is accessed by which module);

In the next section we give an overview about the whole framework, which includes the platform interface, module framework and communication. The automated testing architecture is described in section 3. The Section 4 presents our debug architecture and developed debugging tools followed by conclusion and future work in section 5.

2 Architecture

In oder to integrate different platforms, our project is divided into two parts: platform independent part and platform specific one. The platform independent part is called the *Core*, which can run in any environment. All the algorithms are implemented here. The platform specific part contains code which is applied to the particular platform. In the Core part, several different modules are implemented under the module based architecture. Both parts are connected by a *platform interface*.

Debugging code can be switched on/off during runtime. Debug results are transferred over the network and monitored/visualized using *RobotControl*, a robot data monitoring and debugging tool, which is implemented in Java to be used on arbitrary computer systems (see Fig. 1). It is designed to analyze and debug a single robot, in order to analyze and develop a team behavior of a robot soccer team we need to connect to all the robots at the same time. A related new tool — *TeamControl* is under the development now.

For the implementation we use different programming languages and existing tools and libraries. Here is a rough overview over the used software:

– C++ at the robot
– Java for tools
– premake4 as build system
– glib for communication, etc.
– Google Protocol Buffers (protobuf) for serialization
– Google Test (googletest) and Google Mock (googlemock) for testing

In the following subsections we describe the design and the implementation of the parts mentioned above.

2.1 Platform Interface

Although, the control program should run on real robot or simulation, some other platforms may be useful during the development, e.g., because the real robot is not always available and difficult to manipulate. The physical 3D simulator itself is a very important tool for the development, since it offers easy possibility for testing and debugging. The log-simulator can be used to reproduce a specific situation and trace a bug. Furthermore, in order to isolate the problem and focus on special topic, we don't need the whole robot, e.g., developing new algorithm in computer vision the camera is the only necessary sensor. Therefore, these platforms all play very important roles in the development, so it would be nice to use exactly the same code running in all of them.

In order to do so, we divide the whole program in two parts: the platform independent part (the core or robot control program (RCP)), which contains the actual implementation of the agent (e.g., image processing, motion generation, world modeling, etc.) and the platform specific one, which is responsible for the communication with the particular platform (i.e., getting images from the camera, setting the joint values, etc.). To separate the core from the platform we designed the *Platform Interface* which has to be implemented by each platform specific part for every particular platform. Thus, the platform interface is the middleware between the actual control program (core) and the platform, e.g., real robot, simulator, etc.. It provides all kinds of sensor data to the control program and execute the command received from the control program. The core is only interacting with the platform interface, i.e., from this point of view it cannot differ between different platforms.

At present, our agent (i.e., RCP) can run at 5 different platforms: real *Nao* robot, *Webots* simulator, *SimSpark* simulator, our *log simulator* and the *web cam* simulator. In order to switch between different platforms and run seamlessly on them, the platform interface have to deal with different configurations of sensors and actuators. In this following, we describe the design of the platform interface.

All the sensor data and actuator data are designed as representations (cf. subsection 2.2). For the real robot, representations are motor joint data, image, accelerometer data, gyro data, etc.. For SimSpark, they are joint data, vision data, accelerometer data, gyro data, etc.. For the log-simulator, they are the representations which are recorded in the real robot or simulation. The webcam provides only the image.

From the control program's point of view, the difference between different platforms consists of different representations they support. Therefore, we implemented the unified platform interface, in which the representations can be registered during the initialization if the platform supports.

The 4 platforms are inherited from an virtual unified interface class. All of them implement the register function to create the accessibility to the representations. The control program registers all the necessary representations. As mentioned above, there are some difference between different platforms. Comparing to real Nao robot, some devices are missing in the simulations, such as LEDs and sound speaker. In this case, the platform skips unsupported devices, and uses different modules for different platforms. For example, camera (image sensor) is not used in 3D simulation league. Thus, the controller can disable the image processing module, use the virtual *see* sensor of the simulator SimSpark to provide perceptions. Therefore, the core can run on different platforms and enable different modules for different sensors and actuators.

And also, the control program has to register its main function to the platform, this call back function will be executed by the platform.

The main loop is implemented in the platform interface, the cognition main function is executed after sensing, i.e., the cognition is running after getting the sensor data; the motion main function is executed before acting, i.e., the motion main function has to be done before setting the actuator data.

In the main function implements the classical *sense - think - act loop*, which is typical agent main loop. The multi threads are also supported to run cognition and motion in different frequency, e.g., at the Nao robot the motion runs in 100Hz in the real time thread, and the cognition runs in about 30Hz in another thread.

2.2 Module Framework

Our module framework is based on a *blackboard architecture*. It is used to organize the workflow of the *cognitive/deliberative* part of the program. The framework consists of the following basic components:

Representation objects carrying data, have no complex functionality
Blackboard container (data base) storing representations as information units
Module executable unit, has access to the blackboard (can read and write representations)
Module Manager manage the execution of the modules

A module may *require* a representation, in this case it has a read-only access to it. A module *provides* a representation, if it has a writing access. In our design we consider only sequential execution of the modules, thus the there is no handling for concurrent access to the blackboard necessary, i.e., it can be decided during the compilation time.

We formulate the following requirements on the design of the module framework:

- the modules have no (direct) dependencies between each other (allows to remove or add a module)
- modules exchange information using the blackboard
- the required and provided representations are a *static* properties of a module, i.e., the blackboard is accessed during the construction time of a module
- it is always clear which representations are accessed by a module (i.e., it can be observed during the runtime)
- it is always clear which modules have access to a representation (i.e., it can be observed during the runtime)
- the representations don't have any dependencies required by the framework

In order to provide a simple user interface we use C++ macros to hide the mechanics of the framework. In the following example illustrates the usage of the framework from the point of view of a developer. Here an example for the implementation of a module M which requires the currently seen ball (BallPercept) and provides a very simple model (BallModel) of the ball (just copy the coordinates) :

```
BEGIN_MODULE(M)
  REQUIRE(BallPercept)
  PROVIDE(BallModel)
END_MODULE(M)

class M: public MBase
{
  public:
    M(){}
    ~M(){}

    void execute()
    {
      theBallModell.pos.x = theBallPercept.pos.x;
      theBallModell.pos.y = theBallPercept.pos.y;
    }
};
```

After the registration at the according module manager (one line) this module is executed in the frame of the whole program.

2.3 Communication

Communication between the robot and a PC is essential for debugging purposes and controlling tasks. We partitioned our architecture in a way that the user can choose to use different parts of our software but omit others. Thus we can not assume the availability of some given debugging software GUI (like RobotControl) or the existence of all possible external helper libraries. Even with very few assumptions about the kind of communication and the used software stack we want to give the developers a powerful and flexible communication framework for interacting with the robot.

These thoughts led to the development of the following requirements for our debug communication.

– The communication should be human-readable, e.g. a simple telnet connection can be used to debug the robot. Of course the communication protocol still needs to be easy to understand for computer programs. Self-written or provided debug programs should be supported, but not a necessity.
– We have to be able to transport textual and binary data (e.g. serialized representations).

One important concept of our debug communication is the "command". A command has a name, and several named parameters and their values. A command will always deliver a result. While there are a lot of existing Remote Procedure Call (RPC) libraries around, we did not choose one of these for reasons of simplicity and efficiency. Most of the power of these RPC libraries comes from their type system, especially if they allow to call procedures from different programming languages. Since we want to support different programming languages but still don't want to have the overhead that comes with the different types we choose to only use null-terminated strings for the parameter values and the command result. This approach also matches very well with the requirement to be human readable.

When using strings as the main datatype we have to find ways to represent structured data types or even pure binary data like images. For binary data we propose to use the Base64 encoding. For serialization of structured datatypes different solutions are possible. One possibility is the protobuf library which provides a programming language neutral binary serialization. The encoding and decoding is more efficient compared to non-binary encodings regarding time and space. Other possibilities are more verbose encodings like JSON, YAML or even XML. Our framework does not enforce a specific way of serializing the domain specific data because there seems not to be a solution available that fits well for all different needs. Protobuf is already quite efficient but introduces big dependencies and could be outperformed by manual binary serialization techniques.

In former times the basic protocol for submitting a command and retrieving the result was encoded using protobuf. This approach was abandoned in favor of a more textual protocol that can be easily written by hand but also be generated and parsed by tools like RobotControl.

The EBNF notation for the new command protocol syntax is given below.

```
Command = ["+"], CMDName, {Space, Argument}, "\n";
CMDName = Identifier;

Argument =    StringArg
            | Base64Arg
            | EmptyArg  ;
StringArg = "-", ArgName, Space, ArgValue;
Base64Arg = "+", ArgName, Space, ArgValue;
EmptyArg = ArgName;

Space = " ", {" "};
ArgName = Identifier;
ArgValue = """, {CharNoQuote}, """   | {CharNoSpace};
```

An example for demonstration purposes follows below. This command can be inserted directly on the socket stream e.g. by using the telnet command line program.

```
mycommand −arg1 "value1 with space" −arg2 value2
    argwithemptyvalue +argwithbase64 aGVsbG8= \n
```

Any command begins with its name which is followed by a list of arguments. Arguments can be written directly as a string, encoded using the Base64 algorithm or have an empty value. Prepending "+" to the command name requests the answer to be Base64 encoded. Every command is ends with a line break. The answer will be a null-terminated string.

A separate thread on the robot will listen for incoming commands and will store them in a thread-safe queue. The main thread of the robot can access this queue whenever it wants, and execute the internal command handlers that are connected with the command names. The result of this execution is then put again into an answer queue from where the communication thread periodically sends the messages to the client. The order of the answers will be always the same as the order of the incoming commands and every command will produce an (possible empty) answer.

We implemented a flexible callback register mechanism, that allows us to define a command handler somewhere in the code and register it at the central communication with a certain name and description.

3 Testing

As mentioned in the introduction, the Nao Team Humboldt has a large code base which is being developed by a changing set of team members with various levels of education and knowledge about the project. This leads to various problems with the code, especially when several developers are working on different modules at the same time. Commits to the code repository resulting in non-compiling code in other environments or broken functionality occurred, especially during very busy development cycles during tournament preparations. This slowed down overall progress of the framework.

We hence decided to introduce automated testing in our project. Tests can be run by each developer after code changes to ensure that the changes did not result in undesired side-effects or broken code. The tests are implemented on two different granularity levels:

– Unit-Tests: Calling a specific function with sample input, covering corner cases and invalid input data. Verifies if the results are as expected, and invalid data is being treated correctly with.
– Integration Tests: Tests of a module to ensure that a module follows the requirements of our module framework and works properly. Blackboard-Data is being simulated by a mocking framework - e.g.: The mocked sensor data provides input from the robot standing at one specific position, and the integration test verifies that the self locator results in the correct position.

If a bug is found in the code, a test is being built which triggers the specific issue - once a bug is fixed, the test ensures that it cant be accidentally happen a second time. A very simple test might look like this:

```
TEST(Pose2D, GetAngle) {
    Pose2D nullPose(0,0,0);
    double n = nullPose.getAngle();
    EXPECT_EQ(0, n);
}
```

We use googletest [2] as a xUnit-based testing library and googlemock [1] as a mocking framework. Upon a commit in our code repository, an automated build system compiles and tests the committed source code. If the build or the tests fail, the committer is notified by email about the error. As a side effect, the tests can be used as examples how to interact with more complex functions or modules, which enables new developers to understand the parts of our framework faster and easier.

4 Debug & Tools

In order to develop a complex software for a mobile robot we need possibilities for high level debugging and monitoring (e.g., visualize the posture of the robot or its position on the field). Since we don't exactly know which kind of algorithms will be debugged, there are two aspects very important: accessibility during the runtime and flexibility. The accessibility of the debug construct is realized based on the our communication framework (see 2.3). Thus, they can be accessed during the runtime either directly by telnet or using a visualization software like RobotControl as shown in the Fig. 1). Further, all debug concepts are realized based on the described command executor concept, thus it is easy to introduce new concepts. In contrast to the other parts, the debug infrastructure is not separated from the code, all concepts are statically available and thus they can be used at any position in the code. Some of the ideas were evolved from the GT-Architecture [7]. The following list illustrates some of the debug concepts:

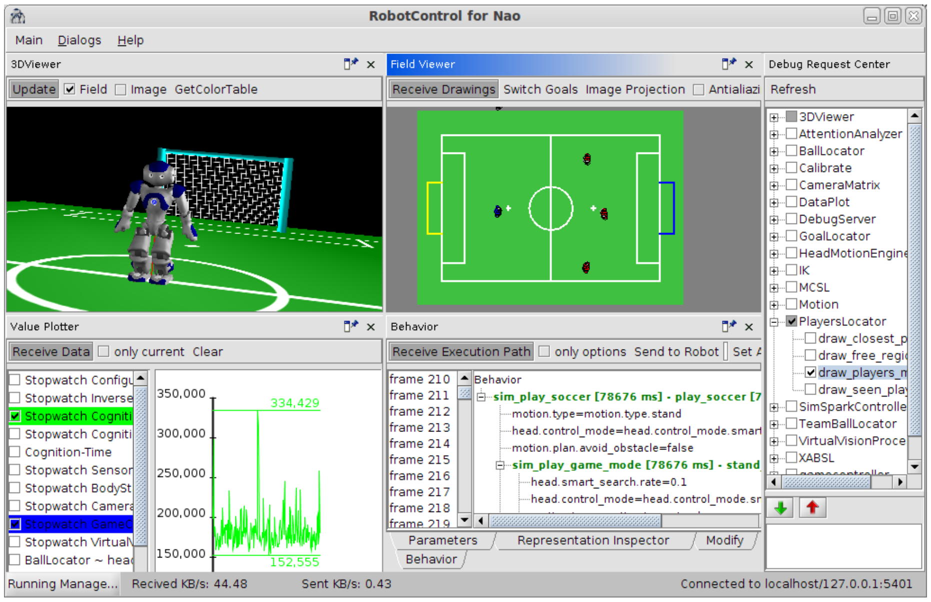
debug request activate/deactivate code parts
modify allows a modification of a value (in particular local variables)
stopwatch measures the execution time
parameter list allows to monitor and to modify lists of the parameters
drawings allows visualization in 2D/3D, thereby it can be drawn into the image or on the field (2D/3D)

As already mentioned, these concepts can be placed at any position in the code and can be accessed during the runtime. Similar to the module architecture, the debug concepts are hidden by macros to allow simple usage and to be able to deactivate the debug code at the compilation time if necessary. Here is an example of usage of the modify macro:

```
void int my_function()
{
  ...
  int x = 5;
  MODIFY("parameter_x", x);
  ...
}
```

Additionally, to these individual debugging possibilities there are some general monitoring possibilities: the whole content of the blackboard, the dependencies between the modules and representations and execution times of each single module.

The Fig. 1 illustrated some of the visualizations of the debug concepts. In particular a field view, a 3D view, behavior tree, plot and the table of debug requests are shown. The TeamControl shown in the Fig. 2 allows for the visualization of the team behavior. In particular the positions of the robots on the field and their intended motion direction are visualized.



```
Behavior
sim_play_soccer [78676 ms] - play_soccer [7
  motion.type=motion.type.stand
  head.control_mode=head.control_mode.smart
  motion.plan.avoid_obstacle=false
  sim_play_game_mode [78676 ms] - stand
    head.smart_search.rate=0.1
    head.control_mode=head.control_mode.sr
```

Fig. 1. The RobotControl program contains different dialogs. In this figure, the left top dialog is the 3D viewer, which is used to visualized the current state of robot; the left bottom dialog plots some data; the middle top dialog draws the field view; the middle bottom shows the behavior tree; and the right one is the debug request dialog which can enable/disable debug functionalities.

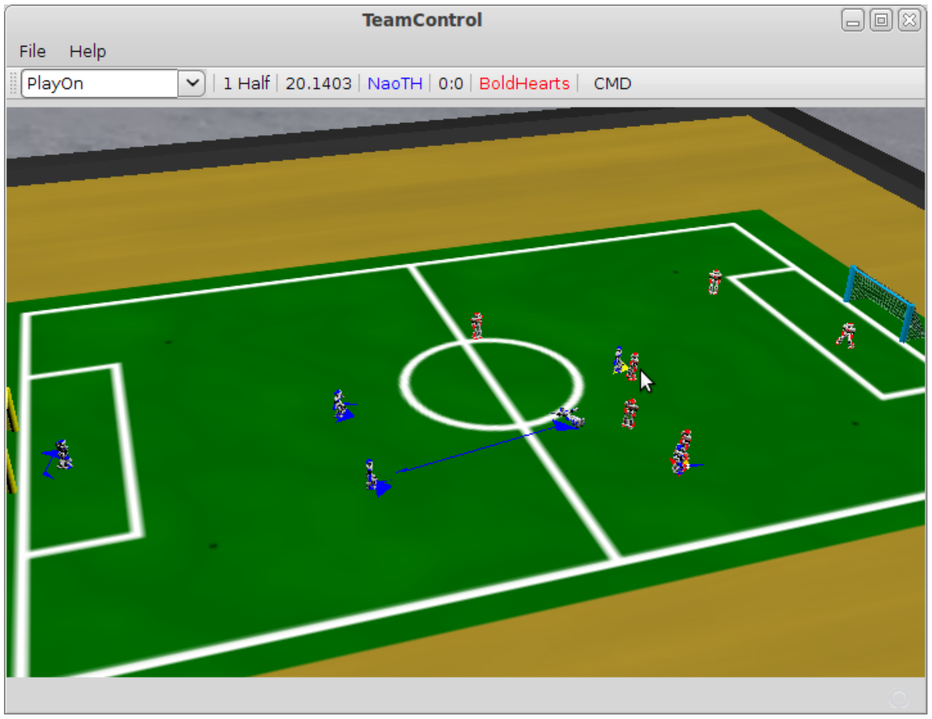


Fig. 2. The *TeamControl* is used to monitor team behavior. In the shown screenshot you can see the positions of the robots on the field and their intended motion direction illustrated by arrows.

5 Conclusion

The presented software architecture is used to drive several robotic platforms including a simulated agent and a real robot Nao. The whole architecture can be divided into several parts platform interface, module framework, communication, debug, tools and tests. Thereby, the main concepts of the overall design are:

– separation between the actual cognitive algorithms and the platform related parts (platform interface);
– separation between time critical part (motion execution) and delay tolerant part (deliberation);
– blackboard based implementation of the deliberation part, i.e., separation of the solutions in modules (which in particular allows parallel development);
– separation between the representations (data) and serialization;
– all components have a very simple user interface, partly realized by macros;

With this architecture we successfully participate (as the only team) in 3D Simulation league and in the SPL at the RoboCup, thereby the programs share the most code. Further, this software is used for the exercising in the AI and robotics lectures at our university.

The next work has to be done on automatic serialization of representations (e.g., generate serialization code during the compilation). The other task is to investigate and to extend the testing framework to the more complicated scenarios (e.g., behavior or probabilistic methods).

Acknowledgments

The authors would like to thank other members of Nao Team Humboldt.

References

1. googlemock - google c++ mocking framework, http://code.google.com/p/googlemock/
2. googletest - google c++ testing framework, http://code.google.com/p/googletest/
3. Bruyninckx, H., Soetens, P., Koninckx, B.: The real-time motion control core of the orocos project. In: IEEE International Conference on Robotics and Automation (ICRA 2003) (2003)
4. Calisi, D., Censi, A., Iocchi, L., Nardi, D.: OpenRDK: A modular framework for robotic software development. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). Nice, France (September 2008), http://openrdk.sourceforge.net
5. Kanehiro, F., Hirukawa, H., Kajita, S.: OpenHRP: Open architecture humanoid robotics platform. International Journal of Robotics Research 23(2), 155–165 (2004)
6. Metta, G., Fitzpatrick, P., Natale, L.: YARP: yet another robot platform. International Journal on Advanced Robotics Systems 3(1), 4348 (2006)
7. Röfer, T., Brose, J., Göhring, D., Jüngel, M., Laue, T., Risler, M.: GermanTeam 2007 - The German national RoboCup team. In: RoboCup 2007: Robot Soccer World Cup XI Preproceedings. RoboCup Federation (2007)
8. Vaughan, R.T., Gerkey, B.P., Howard, A.: On device abstractions for portable, reusable robot code. In: In IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 2421–2427 (2003)