RoboGrams: A Lightweight Message Passing Architecture for RoboCup Soccer

Elizabeth Mamantov[1], William Silver[2], William Dawson[3], and Eric Chown[2]

[1] University of Michigan mamantov@umich.edu
[2] Bowdoin College {wsilver,echown}@bowdoin.edu
[3] Okta wdawson@okta.com

Abstract. RoboGrams is a lightweight and efficient message passing architecture that we designed for the RoboCup domain and that has been successfully used by the Northern Bites SPL team. This unique architecture provides a framework for separating code into strongly decoupled modules, which are combined into configurable dataflow graphs. We present several different architecture types and preexisting message passing implementations, but among all of these, we contend that RoboGrams' features make it particularly well suited for use in RoboCup. As a success story, we describe the Northern Bites' use of RoboGrams and the benefits it has provided to a single team, but we also suggest that it could help SPL teams collaborate in the future.

Keywords: message passing, software architecture, RoboCup, SPL

1 Introduction

The choice of architecture can make or break a software development project. A coherent and well-designed architecture facilitates progress on a system that may have many interacting parts. Conversely, a confusing or unwieldy architecture can hinder improvements to the system. Fitting pieces together awkwardly or forcing code into patterns demanded by a bad architecture can cause frustration for developers and can waste a lot of time.

These statements are particularly true in robotic systems because many different subsystems play a role both in the robot's online processing and in the offline tools needed for development. For example, a typical Standard Platform League (SPL) team will have specialized systems that perform some or all of the following crucial tasks: sensor acquisition, vision, localization, ball location modeling, behavior decisions, motion control, network communication, and data recording. Each of these requires data from one or more of the others, but their methods are not interrelated; it should be possible to develop these modules separately and fit them together in the context of the team's architecture.

This paper presents RoboGrams, a unique message passing software architecture that has been specifically designed to meet these requirements of a robotic system. Its development was inspired by the needs of the Northern Bites, a team that competes in the SPL. After RoboCup 2012, the team decided to convert its code base to a new framework, which presented an opportunity to develop a new software architecture that would be used and evaluated in the RoboCup domain. Since this was a chance to design a unique new framework, we settled on a message passing architecture, which, to our knowledge, is an approach that no other SPL team has attempted. Although there are many existing message passing solutions, we built RoboGrams from the ground up and produced an extremely lightweight and efficient framework with a small dependency footprint. RoboGrams was first used in competition in the 2013 US Open and has been shown to serve the needs of a RoboCup team well. As such, we have released it for the community to use, and the source code can be found online.[1]

2 Background

In an end-to-end robot system, such as a RoboCup soccer player, the agent program must accomplish many different tasks that all rely on the same input data from the robot's sensors and interact in a complex web of data dependencies. As an example of the complexity of data communication in a robot control program, the Northern Bites' system overview is presented in Figure 1. Ideally, each of the tasks will be a black box with known inputs and outputs to any other task that needs its data, and the data-sharing transactions will occur according to clear rules defined by the architecture. Thus, the architecture core needs to provide the "boxes" for processing modules to fill and data "pipes" that connect the modules in a standardized way.

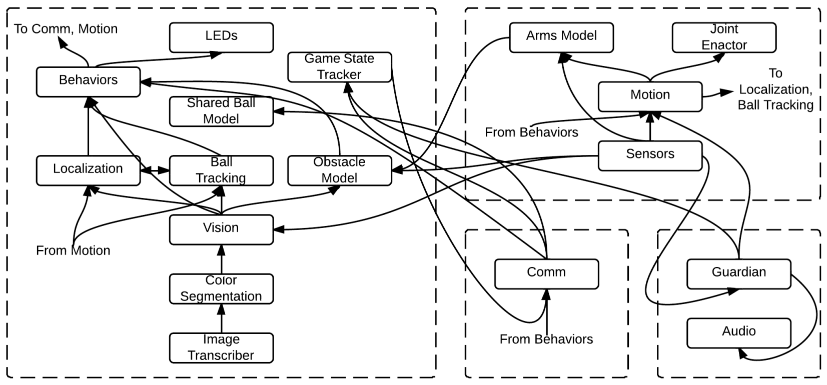


Fig. 1: A dataflow graph of the Northern Bites' online agent program. Separate threads are indicated with dashed boxes.

[1] http://github.com/northern-bites/robograms

The different approaches to solving the problem of providing modularity and data communication result in several common architecture styles seen in the SPL today, which are discussed in the following sections.

2.1 Monolithic

When a team, by default or by decision, does *not* explicitly modularize its code, a monolithic code base results. For many years, the Northern Bites worked within this "non-architecture" framework, in which modules are not black-boxed and interact by calling methods on each other [8]. While this may seem like simply avoiding the architectural design question, it is a valid choice; the main functional systems are still largely separate from each other and data is passed between them. It has the advantages of being both very fast, with no architectural machinery, and very flexible, with no predefined rules.

Unfortunately, these strengths are outweighed by the weaknesses of this approach in the long run. First, it makes adding new functionality difficult because every new system requires a new interface, and with no explicit model of data flow, making a new module interact with existing ones can be tricky. In particular, in a multithreaded environment, setting up each interface individually to avoid data races is an error-prone and difficult process. Second, the lack of explicit black-boxing means that once such an interface is set up, changing one side often requires changing the other; that is, it is hard to update a single processing module without understanding how others work as well. Third, it is impossible to compile and test or use a single component from a monolithic system since everything is tightly coupled to everything else. All in all, a monolithic design may be functionally adequate, but, from the Northern Bites' experience, this setup will limit future team growth and development.

2.2 Blackboard

Blackboard architectures are popular in the SPL and have been studied in other robotic settings as well [11, 14]. In a blackboard architecture, there is a central location for data, known as the "blackboard," and every module reads its inputs from the blackboard and writes its outputs back to the blackboard. This setup unites a robotic system in a coherent fashion, and it provides the desired blackboxing and standardization of data communication. The main reason that blackboards seem to be favored by many SPL teams is that they are efficient; writing to and reading from the blackboard are simple processes, so data movement does not limit the overall system's performance. Nonetheless, this type of architecture has several downsides.

In principle, each module should only be able to access the data it needs as its inputs. With a blackboard, either all of the data is accessible to every module, or bookkeeping must be done to keep track of which modules access which data [3]. That is, a straightforward blackboard architecture does not enforce explicit rules about the visibility of data. Another complication is that a blackboard could be accessed by several threads of execution, so locking mechanisms are necessary for data to be safe. A poorly implemented blackboard will be a hotbed for data races, and a well designed blackboard may require relatively complex locking mechanisms. This issue can be solved by having one blackboard per thread of execution, as in the B-Human framework [11], but then another communication method must be devised and incorporated into the same architecture to transfer information between processes. This design results in a less coherent overall approach. Obviously, many teams have overcome the issues described above, but the key point is that, in practice, a blackboard approach is less elegant than it sounds in theory.

2.3 Message passing

Our chosen approach is a message passing architecture, in which each black-boxed module takes its inputs and produces its outputs in the form of "messages." Rather than keeping all of the messages in a centralized location, messages are passed directly from module to module via input-output connectors. By standardizing the format of a module and specifying how data flow occurs between inputs and outputs, a message passing architecture provides modularity and a standard data communication system, just like a blackboard architecture.

The main argument against message passing in the SPL setting is that it is generally considered less efficient than a blackboard. This is a fair criticism, since heavyweight mechanisms are often employed to move data around a system, making communication costly. However, a message passing architecture lends a robotic system enhanced clarity and structure over blackboards and particularly monolithic systems. With message passing, the setup ensures that the flow of data through the system is made explicit through input-output connections. A message passing system is the perfect software analogy for a dataflow diagram such as Figure 1. It also provides the data-access rules that are lacking in a simple blackboard implementation—modules can only access data for which they explicitly have inputs—and allows for a single, simple solution to the data race problem that is perpetuated by default across all input-output connections. With these features in mind, message passing is a natural and elegant solution to the architecture design problem, provided that the architecture core is implemented efficiently.

3 System Requirements

There were several requirements we adhered to when developing the RoboGrams message passing architecture: speed, modularity, configurability, intra-process organization, and minimal dependencies. Each of these is discussed below.

3.1 Speed

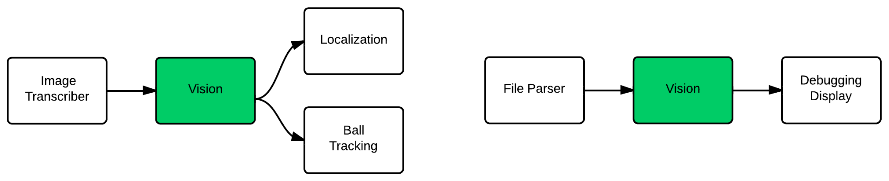
The Nao's camera is capable of capturing images at a rate of 30 per second [1], and our goal was to be able to process each image and run the agent program in real time. A real-time RoboCup system should be capable of handling the full 30 frames per second (fps); any overhead from the software architecture needs to be small enough to not cause a decrease from the maximum frame rate. Given that soccer-playing robots need to function in a dynamic environment in real time, we were not willing to compromise on the speed of the architecture.

3.2 Modularity

The biggest problem with a monolithic and tightly coupled system is that, without black-boxing, modules often have to control or access each others' functionality. Therefore, no single system can be improved or replaced without also editing other modules. For the new architecture, we decided that each module should know nothing about other modules' functionality; the only agreements between two interacting modules would be the format of the data they exchange. Furthermore, no module should need to know where its input data comes from or where its output is sent to because the configuration of the rest of the platform should have no influence on the internal workings of a module.

3.3 Configurability

With the modules completely black-boxed, a related requirement was that the network of modules should be able to be "wired" together in many different ways. Any module that produces output of type T should be able to provide that data to another module which is set up to receive input of type T. As an illustration of the need for this requirement, consider offline vision development, depicted in Figure 2. On a workstation, we need to populate the vision module's inputs with prerecorded images from a file and pass the module's output to a user interface that displays it. The same vision module should be able to run on the robot, except that its inputs would come from a camera module and its outputs would feed into localization and any other system that uses vision data. In a configurable system, the difference between competition, debug, and offline setups is simply a rewiring of most of the same component modules.



(a) Use of a vision module online. (b) Use of the same vision module offline.

Fig. 2: Demonstration of how the same module may be used in two different system configurations.

3.4 Intra-process organization

The Northern Bites and other SPL teams run a robot's agent program as a single, multithreaded process loaded into the Nao's built-in control program, naoqi [2]. Alternatively, the agent program can be broken into multiple processes. Message passing is often used in inter-process communication (IPC), and some SPL teams employ it in this way [11]. However, our goal was *intra*-process communication and organization; it should not be necessary to run every module as its own process to be able to enforce a message passing relationship between them. We thus required that a single process could contain any number of modules, with black-boxing and data communication policies enforced by the architecture core rather than operating system process divisions and IPC.

3.5 Minimal dependencies

This idea applies in two related ways. First, we required that our core architecture depend on as few outside libraries or tools as possible. Not only are outside dependencies often difficult to manage from an administration and setup perspective, but we did not want to make the core of our system dependent on tools we could not completely control. Furthermore, many publicly available libraries have many more features and capabilities than we actually needed, which would have resulted in useless bloat in our platform. The second way this requirement works is within our own platform: each module should have as few dependencies as possible, so the architecture core should be absolutely minimal. In the monolith version of the Northern Bites' platform, compiling any given module necessitated compiling every other related module, so in practice, compiling and running one module required also including the entire platform. With the new architecture, we wanted each module to be dependent only on a small shared core with no extraneous features.

4 Alternative Architectures

As there are many existing message passing architectures, one might wonder if we are reinventing the wheel by implementing our own from scratch. In this section we give an overview of some alternatives and describe why they do not meet our requirements. One obvious option for a robotic message-passing architecture is Robot Operating System (ROS). ROS is an open-source framework that provides the "middleware" to connect various processing "nodes," or modules, via a publish/subscribe communication system [13]. ROS has been tested on the Nao and was proposed for use in the SPL [5]. However, ROS is a large external system that must be painstakingly installed on the Nao and thus does not fulfill our requirement of minimal dependencies. More importantly, it does not provide intra-process organization since every ROS node is a process and the middleware essentially provides IPC. Its performance on the Nao is also uncertain.

One less obvious option that we considered was the Qt framework, which offers signal/slot event-driven connections that could be used to implement data exchange between modules. Qt was developed for use in configurable graphical user interfaces (GUIs) and provides large libraries geared toward GUI development [9]. The core signal/slot functionality, however, can be used for any application, and the Qt core comes pre-installed on the Nao. Nonetheless, our code base would still be dependent on the bulky Qt libraries, so Qt fails our requirement of minimal dependencies. More importantly, without considerable amounts of boilerplate code to enforce system-wide rules, Qt also fails the requirement of modularity. Signals and slots can be linked between modules in such a way that the two modules actually control the timing or flow of each others' processing. We needed our architecture to provide stronger black-boxing.

It is also worth discussing the architectures used by other SPL teams, all of which use blackboards. Among SPL teams, B-Human provides one of the most clearly articulated architectures, and their framework has been adapted for use by several other SPL teams [11, 7]. This architecture was built from the earlier German Team architecture, which successfully integrated the work of several different university teams, a testament to its modularity and the strong organization it provided [10]. The B-Human framework organizes processing into several threads of execution that pass inter-thread messages, but each thread also contains a blackboard. rUNSWift provides a very different overall architecture design, with hierarchical modules that are run in a multithreaded executable [2]. These modules write to and read from a blackboard in order to share information. With yet another alternative blackboard design, Austin Villa separates all robot memory into a "Memory Module" that provides particularly well-structured blackboard functionality for the rest of the team's code [3].

5 The RoboGrams Architecture

With our requirements clearly stated and the need for a novel architecture established, we present RoboGrams, our lightweight message passing architecture. A high-level overview of the different components of RoboGrams is given in Figure 3. It is implemented in C++ and has been tested on Windows and Linux (x86).

5.1 Modules

Modules are the building blocks of a RoboGrams system; each one is a black-boxed unit of processing. To keep the architecture's structure coherent, the abstract class `Module` must be a base of any class that functions as a processing module. To create a derived class of `Module`, one pure virtual function, `run_()`, must be overridden. This function contains the main processing loop of the module. The decision of when to run the module is handled by a higher level of the architecture, so the developer's only concern is providing a module's functionality within the black box of the `Module` interface.

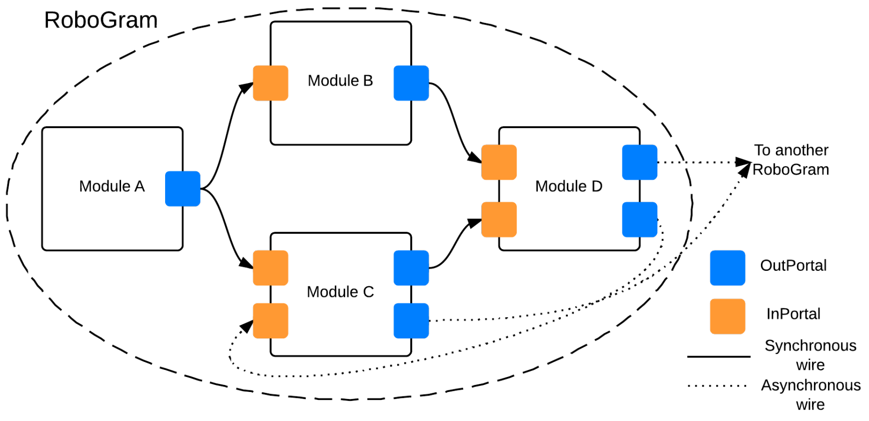


Fig. 3: An overview of the different components of the RoboGrams architecture, with a hypothetical RoboGram composed of several modules.

5.2 Messages

Modules exchange data in the form of the RoboGrams `Message` object. A `Message` is a templated wrapper class that can hold any C++ class that one module needs to pass to another. In the context of the Northern Bites' platform, the contents of a `Message` is typically a Google Protocol Buffer ("protobuf") [6]. Google designed protobufs to be lightweight data containers that also provide serialization and deserialization functionality. Each message type is defined in a `.proto` file, and a provided compiler automatically creates C++ classes to hold and manipulate the specified data types. The Northern Bites platform had already incorporated protobufs for the purpose of logging data and we decided to continue using them for their efficiency and simple interface. However, it should be noted that the messages passed between modules are not serialized nor deserialized in the online system, allowing for extremely fast communication.

The RoboGrams architecture itself, however, is message-type agnostic. Any C++ class can be the contents of a message; for example, protobufs are not ideal for working with images, so we have implemented specialized classes that handle images efficiently. Although using a wrapper class like `Message` rather than simply passing a protobuf or other class directly may be an unintuitive choice, `Message` actually provides a core piece of the architecture's functionality: reference counting.

Reference counting is the mechanism that affords RoboGrams its reliability and efficiency. The main goal of having `Message` count references is so that a copy of a protobuf or other inter-module message class can be made in constant time for each of the modules that needs it. That is, rather than performing a deep copy to ensure that data is safe while it is being accessed by a particular module, all modules simply reference a single existing instance of the data. The number of these references is tracked in `Message` so that the data remains in existence only as long as it is needed—until there are zero remaining references—at which point it can be deleted. By wrapping all of this functionality in `Message`, RoboGrams can take advantage of the benefits of reference counting, but other programmers using the architecture do not have to worry about memory management themselves, thus reducing the likelihood of data races or inefficiencies being written in other parts of the code base.

The reference counting mechanism in `Message` is particularly efficient because of thread-safe atomic operators implemented using inline x86 assembly. Rather than using a mutex to lock critical sections, such as incrementing and decrementing reference counts and copying `Message` pointers, the atomic operators make these actions thread-safe while incurring essentially zero overhead for thread-safety. Furthermore, to make the creation of new messages particularly efficient, `Message` maintains a pre-allocated pool of protobufs—or alternative classes—to draw from when a new instance is needed. This design choice works particularly well when `Message` contains a protobuf, since the Google documentation recommends reusing protobufs in this way [6], but it can be beneficial for other classes as well. Having a pool means that time is not wasted reconstructing objects. All in all, `Message` contains several sophisticated mechanisms that make RoboGrams uniquely efficient, while keeping them wrapped behind a simple interface so that other programmers can use them reliably.

5.3 Portals

Portals are the RoboGrams objects that accomplish the main work of passing messages between modules; a module's data interface is specified by its portals. RoboGrams provides `InPortal` and `OutPortal` objects for modules' inputs and outputs, respectively. Portals must be declared within a `Module`, and their type cannot be changed at runtime. Both `InPortal` and `OutPortal` are template classes that must be provided with a message type, such as a particular protobuf class. `InPortal` provides two key methods, the first of which is `wireTo(OutPortal, bool)`, which allows an `InPortal` to be linked to an `OutPortal` of the same data type. The boolean argument specifies whether this link will be *synchronous* or *asynchronous*; the difference between these options is discussed later in this section. Note that `wireTo` can be called again at runtime, meaning that the wiring configuration can be changed dynamically.

The second `InPortal` method, `latch()`, is used to fetch the message from a connected `OutPortal`. When a message is accessed via `latch()`, its reference count is incremented. The count will automatically decrease when the `latch`ed message goes out of scope, so there is no way for a programmer to forget to release a message back to the pool. The other main functionality that `latch()` provides is that it orders the running of modules so that inputs for a module are produced before that module runs. When a module calls `latch()`, `run()` is called on the connected module if it has not already been run during the current processing cycle so that it can produce updated data.

There is a caveat to this setup, however, since it only works as expected if modules are linked in a directed acyclic graph (DAG) running in a single thread. In fact, in a real RoboCup system, backedges, or links that make loops in the graph, are common, as are links to modules in other threads. We do not want such connections to cause a module to run since backedges would cause infinite run loops and requests from another thread would cause data races. Thus, these two types of edges must be marked asynchronous when `wireTo` is used to make the connections. In sum, `latch()` on synchronous edges causes input modules to run, while on asynchronous edges, it simply accesses existing data.

5.4 RoboGrams

RoboGrams provides an additional, higher organizational level than most message passing systems: the `RoboGram`, a shortening of "RoboCup Wiring Diagram." As its name suggests, `RoboGram` is a collection of `Modules` that are typically wired together in specifying C++ code and run at the same rate. `RoboGram` simply holds a list of `Module`s and calls `run()` on each of them. Due to the way that `latch()` works (see previous section), the modules are automatically processed in topological order; data dependencies are produced before running the module that requires them. Each module is run just once per diagram cycle.

Each `RoboGram` structure is its own thread of execution that can run asynchronously with any other `RoboGram`s in the same system. That is, each diagram maps naturally onto a thread in a multithreaded process. In the Northern Bites' system, for example, there are four `RoboGram`s, each running in its own pthread thread at different rates. For example, the motion diagram runs at 100 Hz to synchronize with the Nao's DCM, whereas the cognition diagram, which contains vision, world modeling, and behaviors, runs at 30 Hz to synchronize with the Nao's camera.

6 Impact

The RoboGrams architecture has proven to be an overwhelming success based on the Northern Bites' experience using it for RoboCup development. The team's code base has been decluttered and streamlined, and no sacrifices were made in terms of runtime. The most obvious difference facilitated by the new architecture is a dramatic improvement in code quality and readability, accompanied by a drop in overall lines of code in the code base. That is, from our experience, RoboGrams can improve a team's development environment.

Legacy code is a significant challenge for any RoboCup team with a long history, so one major benefit of the modular nature of RoboGrams is that is leads to a more forgiving learning curve for new team members. In a truly modular code base, a new member can focus on one module and make valuable contributions to the team relatively quickly. This is a particularly important consideration for teams composed mostly or only of undergraduates, including the Northern Bites, where new team members may have little programming experience and membership cycles quickly as students graduate.

Beyond the striking simplification, RoboGrams can also facilitate the addition of new functionality and support an improved testing environment. Because RoboGrams modules can be tested individually, the Northern Bites were able to build and make use of a continuous integration framework to replace our previous, less rigorous testing methods [4]. Also, the barrier to adding new modules in a RoboGrams-based system is extremely low; the team was able to add a module for obstacle modeling in just a few days, which resulted in noticeable improvements to our robots' competition play.

While all of the above improvements are valuable, RoboGrams would have been unusable if it added significant time overhead to a system; for use in RoboCup, the architecture absolutely needed to be capable of handling a real-time agent program. Fortunately, the pains taken to make our message passing implementation extremely efficient paid off. Based on a recent set of five time trials, the Northern Bites' current average frame rate is 29.31 fps, and there was no significant difference in frame rate between this code, which actually includes several new processing modules, and pre-RoboGrams code. This relationship strongly suggests that the bottleneck on our processing is the Nao's camera frame rate, not the RoboGrams architecture.

7 Conclusion

We have presented RoboGrams, an elegant and lightweight architecture for robotic systems that fills a unique niche among existing software architecture styles and has been very successfully applied in the SPL domain. Like all of the software developed by the Northern Bites, RoboGrams is open-source and is freely available online. While the new architecture's positive impact on the Northern Bites alone has been dramatic, we believe that sharing it among other SPL teams could result in increased collaboration and league-wide progress.

Since all SPL teams are attacking the same research problem and using the same hardware, in some sense, the SPL is one big software development project. Ideally, any team could borrow another's vision module, for example, and swap it into their own system for testing or try to improve the original. To keep the barrier to entry reasonably low for new teams, this sort of sharing—particularly of walking engines—is crucial. However, in practice, it is usually a struggle to separate one component of a team's code base, and teams often end up using more of another team's code than they need.

Due to its flexibility and simplicity, RoboGrams can enable this type of true collaboration. Each RoboGrams module is strongly decoupled from the rest of a given team's system and can be easily adopted for use by another team—even if that team's code is not based on RoboGrams. To make use of a module is simple: the correct data format must be supplied to its inputs. Thus a team can take advantage of the higher diagram organization, or not; in either case, RoboGrams facilitates sharing. Furthermore, since the architecture can run in a process with a single thread, a multithreaded process, or even be extended to work in multiple processes via IPC, any team's setup will be RoboGrams-compatible.

An initial set of RoboGrams modules are provided open-source in the Northern Bites' code base.[2] These include a module version of B-Human's 2011 walking engine [12]. While RoboGrams has been invaluable to the Northern Bites and is clearly already a success, we hope that it can also help spark greater SPL-wide collaboration in the future. We thus share RoboGrams first as an innovative and elegant take on a robotic software architecture and second as a tool that the rest of the SPL and RoboCup as a whole is invited to utilize.

References

1. Aldebaran: Video camera. https://community.aldebaran-robotics.com/doc/1-14/family/robots/video_robot.html#robot-video, accessed February 2014
2. Ashar, J., Claridge, D., Hall, B., Hengst, B., Nguyen, H., Pagnucco, M., Ratter, A., Robinson, S., Sammut, C., Vance, B., et al.: RoboCup Standard Platform League—rUNSWift 2010. In: Australasian Conference on Robotics and Automation (2010)
3. Barrett, S., Genter, K., He, Y., Hester, T., Khandelwal, P., Menashe, J., Stone, P.: UT Austin Villa 2012: Standard Platform League world champions. In: RoboCup 2012: Robot Soccer World Cup XVI, pp. 36–47. Springer (2013)
4. Dawson, W.J.: Extensible Continuous Integration Framework. Undergraduate honors thesis, Bowdoin College, Brunswick, ME (2013)
5. Forero, L.L., Yanez, J.M., del Solar, J.R.: Integration of the ROS Framework in Soccer Robotics: the NAO Case. In: RoboCup 2013: Robot Soccer World Cup XVII. Lecture Notes in Artificial Intelligence, Springer (2014)
6. Google: Protocol buffer developer guide. https://developers.google.com/protocol-buffers/docs/overview, accessed February 2014
7. Hofmann, M., Schwarz, I., Urbann, O.: Nao Devils Team Report (2013)
8. Neamtu, O., Dawson, W., Googins, E., Jacobel, B., Mamantov, E., McAvoy, D., Mende, B., Merritt, N., Ratner, E., Terman, N., et al.: Northern Bites code release (2012)
9. Qt: Qt Project. http://qt-project.org/, accessed February 2014
10. Röfer, T.: An Architecture for a National RoboCup Team. In: RoboCup 2002: Robot Soccer World Cup VI. pp. 417–425. Springer (2003)
11. Röfer, T., Laue, T.: On B-Human's Code Releases in the Standard Platform League - Software Architecture and Impact. In: RoboCup 2013: Robot Soccer World Cup XVII. Lecture Notes in Artificial Intelligence, Springer (2014)
12. Röfer, T., Laue, T., Müller, J., Fabisch, A., Feldpausch, F., Gillmann, K., Graf, C., de Haas, T.J., Härtl, A., Humann, A., Honsel, D., Kastner, P., Kastner, T., Könemann, C., Markowsky, B., Riemann, O.J.L., Wenk, F.: B-Human Team Report and Code Release 2011 (2011), only available online: http://www.b-human.de/downloads/bhuman11_coderelease.pdf
13. ROS: Communications Infrastructure. http://www.ros.org/core-components/#communications_infrastructure, accessed February 2014
14. Tzafestas, S., Tzafestas, E.: The Blackboard Architecture in Knowledge-Based Robotic Systems. In: Jordanides, T., Torby, B. (eds.) Expert Systems and Robotics, NATO ASI Series, vol. 71, pp. 285–317. Springer Berlin Heidelberg (1991)

[2] http://github.com/northern-bites/nbites