# Design Principles of the Component-Based Robot Software Framework Fawkes

Tim Niemueller[1], Alexander Ferrein[2], Daniel Beck[1], and Gerhard Lakemeyer[1]

[1] Knowledge-based Systems Group
RWTH Aachen University, Aachen, Germany
{niemueller, beck, gerhard}@kbsg.rwth-aachen.de
[2] Robotics and Agents Research Lab
University of Cape Town, Cape Town, South Africa
alexander.ferrein@uct.ac.za

**Abstract.** The idea of component-based software engineering was proposed more that 40 years ago, yet only few robotics software frameworks follow these ideas. The main problem with robotics software usually is that it runs on a particular platform and transferring source code to another platform is crucial. In this paper, we present our software framework Fawkes which follows the component-based software design paradigm by featuring a clear component concept with well-defined communication interfaces. We deployed Fawkes on several different robot platforms ranging from service robots to biped soccer robots. Following the component concept with clearly defined communication interfaces shows great benefit when porting robot software from one robot to the other. Fawkes comes with a number of useful plugins for tasks like timing, logging, data visualization, software configuration, and even high-level decision making. These make it particularly easy to create and to debug productive code, shortening the typical development cycle for robot software.

## 1 Introduction

The idea of component-based software engineering (CBSE) dates back more than 40 years with McIlroy's demand to create reusable software entities [1]. It follows three main ideas: (1) develop software from pre-produced parts, (2) reuse software in other applications, and (3) be able to maintain and customize parts of the software to produce new functions and features [2].

In [3], Brugali et al. discuss current trends in component-based robotic software. Component-based robotic software deals with applying the principles of component-based software engineering to robotic control software. Following [4], they require four properties to be fulfilled for a robot software component suiting the component-based approach: (1) a component is a binary (non-source-code) unit of deployment, (2) a component implements (one or more) well-defined interfaces, (3) a component provides access to an inter-related set of functionalities, and (4) a component may have its behavior customized in well-defined manners without access to the source code. In [3], the authors conclude that in a component-based robotic software (CBRS) system, the used component model allows for observing and controlling the internal behavior of the component, and that each component has well-defined *interfaces* and *data structures* from which the usage modalities follow. Moreover, a CBRS requires *communication patterns* that enables the *interconnection of reusable components*, while the level of abstraction of the communication infrastructure must be such that *heterogeneous components* can be connected. Finally, they demand for *component repositories* which simplify documentation, retrieval, and deployment of a large number of reusable components. Examples for systems which, at least, partially fulfill these requirements, as stated by the authors are, for instance, [5–8].

In this paper, we propose the Fawkes robot software framework (RSF) which follows the component-based approach. The aim is to provide a software environment which cuts short the development times for robot software components. Our target platforms are wheeled service robots as well as humanoid soccer robots. After an intensive study of available RSFs, we saw the need to develop a component-based framework which could be used cross-platform for all of our robots. Even frameworks like ORCA [8, 9] which follow the same software paradigm, did not meet our demands, as we lay out below.

The key features of Fawkes are: (1) a well-defined component concept, (2) a hybrid blackboard/messaging infrastructure for communication, (3) well-defined interfaces, (4) run-time loadable plugin mechanisms inheriting certain predefined aspects such as communication, configuration, logging, timing or integrating computer vision and networking (following the aspect-oriented software design), (5) utilizing multi-core computation facilities by deploying POSIX threads for plugins, and (6) a network infrastructure for communicating with remote software entities.

In the following, we present the Fawkes software framework. In particular, we review recent existing approaches and related work in Section 2, before we state in Section 3, what the design criteria are that we based our work on. In Section 4 we describe the concepts of Fawkes in detail. We present the general interface structure, the communication middle-ware, software engineering concepts such as guarantees, and show how the main application can be distributed. We want to remark that Fawkes comes with a behavior engine [10], but is not limited to the built-in one. In Section 5 we present the application of Fawkes on our different robot systems. We conclude with Section 6.

## 2 Other Robot Software Frameworks

Several other robotics frameworks and middle-wares for robots exist. Here, we want to give a brief overview of the most recent and related ones. Amongst them, the Open Robot Control Software [11], Orocos for short, is driven by the desire to standardize robotics software components and especially their interconnection. The aim is to provide a comprehensive framework for robotics applications. For communication, Orocos deploys CORBA. Several libraries for robotics applications are available for Orocos. The basic framework is defined in the Real-Time Toolkit, further, there are libraries for Bayesian filtering, robot kinematics and dynamics. A spin-off of Orocos is Orca [9]. Orca provides a communication infrastructure that uses ICE instead of CORBA, which is a successor development of CORBA with a simplified API and where redundant features were left out. Orca comes with drivers for a wide range of common hardware and solutions for basic robotics problems such as path planning. Software components are run as different processes which communicate with each other. However, it seems that Orca as a framework is lacking coherence, as the overall design is only loosely defined and fragmented into several pieces. The main benefits of using a CORBA-based middle-ware is the possibility to use a variety of programming languages and the ability to distribute computations over many hosts transparently. For mobile robots, however, most if not not all of the computation has to be done on the robot. Therefore it does not seem desirable to have a purely network-based communication framework for these application domains. Moreover, some general criticism on the use of CORBA states [12]: "First and foremost the API is complex, inconsistent, and downright arcane and writing any non-trivial CORBA application [is] surprisingly difficult." Other technologies such as SOAP or XML-RPC are used as replacements, especially because of their simplicity, but they are slightly less efficient.

Another recent and popular example for a robot software framework is the Robot Operating System (ROS) [13]. The idea behind ROS was to develop a framework for large-scale service robot applications. ROS offers a communication infrastructure which for one employs peer-to-peer messaging and for the other provides remote procedure calls (RPC). It uses a central broker for service registration and discovery. For one ROS processes, called nodes, communicate directly with each other by exchanging messages following a topic-based publisher/subscriber philosophy. In contrast to this, so-called service communication allows for strict request and answer message passing. In these messages, primitive data types and structures are allowed which can be composed to more complex messages. Communication with the broker and between nodes for connection negotiation employs XML-RPC. Topic communication and RPC is implemented as a custom protocol over TCP or UDP. Besides the basic communication services, ROS offers specialized packages such as predefined message ontologies (e.g. for navigation or sensor data), or data filters. To facilitate the programming with ROS, APIs for C++ and Python are offered. For ROS, a large number of third-party libraries for standard tasks such as localization or navigation is available. Although the ROS approach looks promising and seems to be very general, the different design criteria suggest Fawkes as a viable alternative. Most notably, ROS conveys systems of only loosely federated nodes, while Fawkes emphasis es a closely integrated system. This is beneficial for synchronization of different tasks, low latencies, and efficiency from embedded systems to multi-machine service robots. ROS and Fawkes share the idea of a component-based design, but in Fawkes this is set as a priority, e.g. by emphasizing re-use of well-known interfaces as much as possible, while in ROS it is common for each module to define its own message types.

## 3 Terminology and Design Criteria

In this section we present the design criteria for Fawkes. As our conceptual design follows the component-based software paradigm, we start with introducing the terminology used by this paradigm in Sect. 3.1, before we review some characteristics of robot software frameworks in Sect. 3.2. Finally, in Sect. 3.3 we overview our design goals.

### 3.1 Terminology

According to Szyperski [2], different forms of reuse are possible at the design level. He distinguishes between (1) sharing programming/scripting languages, (2) libraries, (3) interfaces, (4) messages/protocols, (5) patterns, (6) frameworks, and (7) system architectures. In the following, we briefly introduce the basic terms and address the different sharing options.

A *component* is defined as a binary unit of deployment that implements one or more well-defined interfaces to provide access to an inter-related set of functionalities, configurable without access to the source code [8]. It adheres to a specified "contract" and expects certain input data and produces and provides specified output data. The contract also states what the component expects from its context, e.g. certain timing constraints. If the requirements posed by a certain system infrastructure are not met by an interface, it has to be wrapped accordingly. A *module or library* is a coherent set of implemented functionalities, using an object-oriented data encapsulation as a useful (but not necessary) design paradigm. A module can be compiled separately, and is portable to different platforms given compatible compiler and operating system support. Modules inherently hardware-dependent, such as device drivers, are often not portable [9]. The *system architecture* is a specific choice of functional building blocks (*components*), in order to build a software system that performs according to a specification [14].[2] basically distinguishes between (strict) layered architectures and the (strict) onion model. The disadvantage of strict layered architectures is that the extensibility of the system is restricted. A *framework* is a design and an implementation providing a possible solution in a specific problem domain. It is used to model a particular domain or an important aspect thereof. Frameworks are similar to software libraries, which means they are reusable abstractions of code wrapped in a well-defined API. Unlike libraries, however, the overall program's flow of control is not dictated by the caller, but by the framework. This inversion of control is a distinguishing feature of software frameworks [15]. Finally, a *component architecture* determines the internal design of one single component in order to guarantee that the component performs according to its external interface/contract.

### 3.2 Characteristics of a Mobile Robot Software Framework

There exists a plethora of different robot frameworks today. On the one hand, different approaches how a robot should be controlled emerged and on the other hand, different methods how modules are combined and connected were developed during the last decades. In [16], certain characteristics are defined for the evaluation of software frameworks that we summarize and extend below:

– *Robot hardware abstraction.* The framework should not be tailor-made for a specific robot platform, but it should rather be portable to a variety of platforms.
– *Extensibility and scalability.* The robot framework must be able to easily incorporate new software modules and to use hardware newly added to the robot.
– *Limited run-time overhead.* The run-time overhead can be measured in terms of memory and CPU requirements, i.e. the frequency by which the control loops are executed, and end-to-end latency, meaning the time that is required for a sensor reading to have an effect on the actuator command.
– *Actuator control model.* The actuator control model is twofold. For one, the overall structure of a robot software system imposes a certain preferred model of control (deliberative vs. reactive), for another different actuator control models must be available depending on the type of actuator.
– *Software characteristics.* Software for mobile robots shares the common requirements for good software, like completeness, simplicity, correctness, and consistency.
– *Tools and methods.* The complexity of robotics tasks demands for data introspection, monitoring, or debugging tools provided by the framework in order to allow efficient software development.
– *Documentation.* To achieve wide acceptance for a software platform rigorous documentation is crucial and has to be provided in forms of reference manuals and API documentation.

### 3.3 Design Goals of Fawkes

Inspired by the component-based software design paradigm, with Fawkes we tried to combine these with the requirements of a robot software framework. Moreover, we were influenced by our personal experience of having programmed robots in service robotics and robotic soccer domains for nearly a decade. Our goal was to design a software framework that is as flexible as possible and portable to our various robot platforms ranging from domestic service robots to biped soccer robots. Across the different platforms a consistent environment must be provided. The software must scale from embedded systems to multi-machine robotic applications. The number of concepts that a user needs to know should be as few as possible to minimize the overhead to get familiar with the framework. Finally, the run-time overhead needed to be as small as possible as we wanted to deploy Fawkes in real-time domains such as robotic soccer. The framework must be extensible to add new functionality over time.

In a nutshell, the framework has the following features. Each functional entity in Fawkes is a component, which can make use of several predefined aspects. Each component, implemented as a plugin that can be loaded at run-time, needs to inherit a communication aspect to communicate with other components. Plugins are realized as threads in a monolithic program. However, distributed design is possible by synchronizing the data between multiple instances via a network protocol with a minimal timing overhead. Via synchronization among the components we ensure that no superfluous work is performed.

## 4 The Fawkes Robot Software Framework

In the following, we describe the Fawkes framework in detail. In order to do so, we start by showing the general structure of a Fawkes application in Sect. 4.1, and derive from there the different properties which qualifies Fawkes to be a component-based design. In particular in Section 4.2, we show the interface design and the communication infrastructure. Section 4.3 overviews the predefined software patterns that comes with Fawkes, so-called aspects. These predefined aspects allow for introducing the concept of guarantees, which provide some basic guarantees of the quality of service of the application.

### 4.1 The Framework Architecture

Following the component-based approach, we define components in Fawkes as logical elements. They manifest in the form of plugins. Generally, a single plugin implements one component.[3] The Fawkes core application only provides the basic infrastructure. The most important elements of that infrastructure are a central blackboard used for exchanging data between plugins, centralized logging facilities that allow for parallel logging to multiple logging targets (i.e., console output, log-files, etc.), and a centralized configuration database that is intended to store the configuration parameters for all components of the system. Moreover, it provides a default implementation of a main loop that might be replaced with a custom main loop at run-time. Generally, the main loop controls the execution order of the threads and ensures that certain timing criteria are met. Each run of the main loop is fractured into multiple stages; the default implementation represents a refined *sense-think-act* cycle: in the first stages in the loop the threads acquiring new data from the robot's sensors are run, afterwards the threads implementing deliberative or reactive decision-making components, and lastly the threads which send commands to the actual hardware.

The actual functionality that makes an arbitrary framework a robot software system is provided by plugins. The plugins are implemented as dynamically loadable libraries—shared objects on Linux systems. They implement a particular interface which gives access to descriptive and dependency information and a set of threads. Plugins can be loaded and unloaded at run-time. This allows for a fast development cycle. Usually a developer works on one plugin at a time. With the ability to reload only this plugin the program-compile-test cycle can

[3] There are situations where it is useful to combine multiple components into a single plugin for efficiency or direct synchronization, which is supported by the framework.

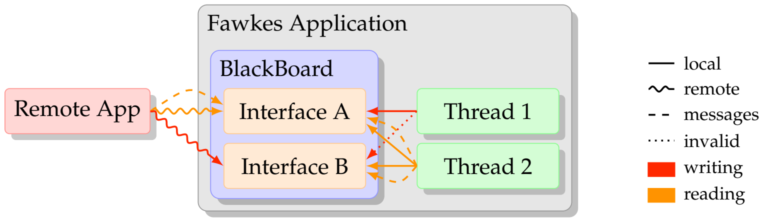

**Fig. 1.** Accessing the Fawkes blackboard.

be quicker, because only the changed plugin has to be reloaded, not the whole system.

Threads are one of the key elements of the Fawkes RSF. With the advent of modern multi-core CPUs it is considerably worthwhile to provide simple ways to exploit the multi-processing capabilities. With the decision to make every functional entity of plugins a thread, it is reasonably easy to exploit this feature. Threading is implemented based on the POSIX Threads API. They can operate in two different modes, either in *continuous* or in *wait-for-wakeup* mode. In the continuous mode, the thread runs all the time, until it exits or is terminated by another thread. In the wait-for-wakeup mode, the thread blocks, until it is woken up. When woken up, it executes a single iteration. Precisely, a plugin in wait-for-wakeup mode registers for a certain stage provided by the main loop. When all threads of the current stage in the main loop have finished their iteration (or if certain timing constraints are not met, i.e., a plugin runs longer than it is supposed to) the main loop proceeds to the next stage and wakes up all plugins registered for that stage. Note, the plugins registered for the same stage in the main loop run concurrently.

## 4.2 Interface Design and Communication Infrastructure

As already mentioned above the Fawkes core provides a blackboard that serves as a centralized storage unit that plugins can read data from and write data to (cf. Fig 1). The access to the data stored in the blackboard is managed using instances of interfaces whose types are known to all plugins. An interface defines a number of fields; the definition of each field consists of a basic data type and an unique identifier. For instance, an interface of type *Position2D* may contain fields of type *float* with the names $x$ and $y$. To facilitate the design, tool support for creating and maintaining interfaces exists.

The plugins may request to open an interface of a certain type with a certain (unique) identifier at run-time. Additionally, the plugin may either open the instance as a writer or as a reader. Whereas multiple readers may attach to an instance of an interface, only a single writer is allowed to do so. This ensures the integrity of the data stored in the blackboard. With saying a plugin "opens an interface", we actually mean that it is first checked whether an instance of that type of interface with the given identifier already exists in the blackboard. In that case, a proxy of the interface in the blackboard, i.e. a local copy (outside of the blackboard), is created for the plugin. In case no instance of an interface conforming to the request of the plugin exists the necessary memory is allocated by the blackboard first. The plugins can synchronize their proxies with the original in the blackboard either by copying the current data from the blackboard to the proxy or the other way around.

This reader-writer model allows to pass data from the writer to the reader. Readers may send commands to the writer of an interface instance by means of messages. Associated with each writing instance of an interface is a message queue that stores the messages sent by the readers in the order they have been sent and thus allows the writer to process the messages in the correct chronological order. Although neglected above, a definition of message types accepted by an interface is also part of the interface definition besides the definition of the data fields provided by an interface.

### 4.3 Aspects and Soft Guarantees

Plugins need to access features provided by the Fawkes framework, for instance, accessing the blackboard. In order to reduce the implementation effort we borrowed ideas from aspect-oriented programming. A so-called *aspect* denotes a specific ability or requirement. Now, when a thread of a plugin wants to make use of such an ability, it is "given that aspect". In our C++ implementation this means that we have a class implementing each aspect; a thread "is given an aspect" by inheriting from that class. In a sense, we lift the classic aspect-oriented programming paradigm onto the framework level and aspects assert concerns of threads regarding particular framework functions like centralized logging.

The framework allows to specify dependencies between different aspects. For instance, there is the *vision-master* aspect that is given to threads that provide access to cameras; the *vision* aspect is given to threads that need to access the images captured by the cameras. A modeled one-to-many dependency guarantees that a plugin which has at least one thread with a *vision* aspect can only be started when another having the *vision-master* aspect is already running; as long as at least one thread with a *vision* aspect is running it is not allowed to unload the plugin which owns the thread having the *vision-master* aspect.

Besides the *vision-master*- and *vision* aspect, the Fawkes framework implements aspects (among others) which allow to access the blackboard, the central configuration database, and the centralized logging facilities. Furthermore, there is a central clock in the Fawkes framework (accessible via the *clock* aspect). The *time-source* aspect allows threads to provide a proprietary clock which especially comes in handy when working with a simulation which might not run at real-time. In such a case a thread with the *time-source* aspect can provide the simulation time to all threads that have the *clock* aspect via the central clock.

Above we already mentioned a couple of (soft) guarantees. The idea behind those guarantees is that they provide a simple exception handling mechanism on the framework level. For example with the framework knowing the requirements of threads because of its aspects it can ensure that all requirements and dependencies are fulfilled. These guarantees are called “soft” because they are not checked all the time but only at certain moments. Dependencies, for instance, are only checked during initialization/finalization of the respective threads but not in between.

## 5 Evaluation and Case Studies

In this section evaluate the framework by the characteristics provided in Section 3.2 and show two configurations of Fawkes in different application domains. We start with our service robots and show how legacy software from our previous software framework is intertwined with Fawkes. When integrating legacy software, the component-based approach clearly pays off. The second domain is the robotic soccer domain. In this domain we participate in RoboCup competitions in the Middle-size league with wheeled robots, and in the Standard Platform League with the biped humanoid robot Nao.

### 5.1 Evaluation of Framework Characteristics of Fawkes

As mentioned in Section 3.2, certain characteristics can be used for a qualitative evaluation of RSFs. In this section we apply these to Fawkes.

*Hardware abstraction* is accomplished by encapsulating hardware-specific functions into separate plugins. Data is shared and commands are sent via the blackboard. Therefore, adapting to a new platform is done by implementing and loading the set of plugins that matches the chosen hardware. *Extensibility* is accomplished by the component-based approach. This makes it easy to use existing components and add new ones. For Fawkes *scalability* is especially targeted towards singular machines assuming all computation is done on the robot. Since Fawkes makes use of multi-threading at its core, there is a high potential for exploiting today’s multi-core machines. Experiments suggest that the *run-time overhead* of the base system in relation to the functional plugins can be considered negligible. High frequency main loop iterations are achieved as long as the individual threads are bounded appropriately in time. In the future, a closer comparison to ROS might be performed. The *actuator control model* that is implemented by a plugin can be freely chosen. For the overall framework the hybrid deliberative-reactive paradigm is presumed. *Tool support* is very important when developing software in general, and for complex systems like a robot in particular. We have carefully designed the software to interact well with debuggers and performance analysis tools. *Documentation* of all public APIs is enforced in the development process. A wiki provides usage and developer documentation.

The Fawkes RSF is complete in the sense that it provides the *basic infrastructure to implement and interconnect a set of components* to control a certain robot. For a particular platform and domain the applicable components must be developed and integrated. The *correctness* is not automatically verified. For several parts of the software quality assurance applications have been written, that can be used to manually test parts of the software. To achieve *consistency* functional blocks have been bundled into appropriate libraries and well-known interfaces are used for communication wherever feasible.

### 5.2 Service Robots

Our service robot employs a differential drive; it is equipped with 360° laser range finder, a 6 DOF robotic arm, and a stereo camera. Additionally to two computers in the base (900 MHz Pentium III) which handle localization, navigation, and processing of the laser distance readings we added another computer (2 GHz Core 2 Duo) that runs vision applications (object and face recognition), speech recognition, people tracker, and the control software for the robotic arm and the pan-tilt unit on which the stereo camera is mounted. As such this robot is a multi-node system on which a multitude of (partially) quite demanding applications is run.

Fawkes runs distributed on the three machines, and data and commands are transferred between the machines using the remote blackboard mechanisms provided by the framework. A specialty is that we integrated several applications developed for our old framework into the system (localization and navigation). This could be easily accomplished by extending the “old” applications by means of small adapters to exchange data and commands with Fawkes applications, again, using the remote blackboard mechanism. This shows that third-party software can be easily integrated into Fawkes and that even two frameworks that build on different design concepts can be used side-by-side. It has to be noted, though, that applications which are integrated in such a way are not equivalent to Fawkes plugins as their execution is not synchronized via the main loop, external applications cannot be given any aspects, and guarantees cannot be made for such external applications. These are no limitations of the Fawkes framework but a tribute one needs to pay due to different design principles underlying different frameworks. In our case, for example, the applications from the old framework handled their timing on their own. Consequently, it is not possible to synchronize them with the other plugins without major modifications which basically coincide with re-implementing the old applications as Fawkes plugins.

### 5.3 Humanoid and Wheeled Soccer Robots

Figure 2 gives an overview of the components and interfaces that are running on our wheeled soccer robots. On the left-hand side of the figure, hardware components can be found such as driver for the kicker mechanism (*kicker*), the motors (*navigator*), or the camera (*fvbase*). The kicker serves the kicking interface, while the navigation component fills the odometry and the navigation interfaces. The camera drivers (we have two camera systems mounted on the platform) provide images in a special interface. These images are used by several medium-level components, such as the localization with the installed omni-visual camera (*omni-loc*), the ball detection (*omni-ball*), and the obstacle detection (*omni-field* and *stereo-obs*). All these modules feed their particular interfaces

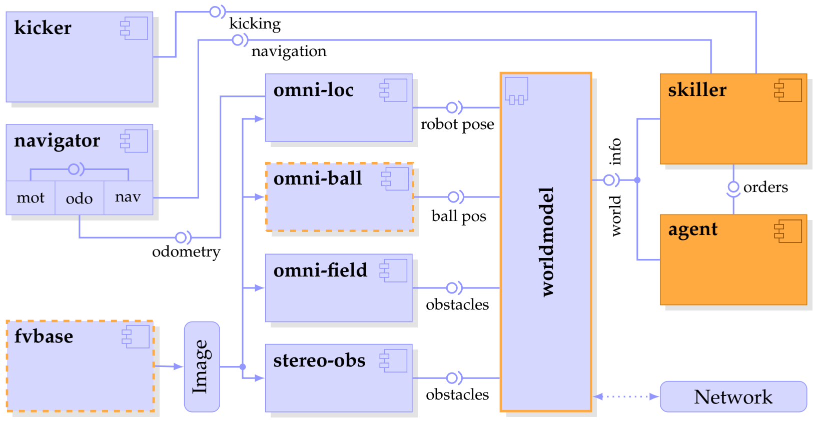

**Fig. 2.** Fawkes components of our wheeled soccer robot

which are amalgamated into a consistent world model. Note that the world model also has a link to the robot's network to integrate data from other robots. The world model in turn feeds the *world info* interfaces which is used by the high-level decision making (the *skiller* to execute and monitor basic actions, and the *agent* to take decisions which action to perform). The architecture layout for our biped soccer robot Nao looks similar, although the platform is very different.

## 6 Conclusion

In this paper, we presented the Fawkes RSF and its design principles, which follow the component-based software paradigm. This paradigm turned out to be very beneficial, as we applied Fawkes to several of our robot platforms and the component-based approach facilitated this endeavor. Our evaluation and the examples in Sect. 5 suggest that the framework meets the requirements outlined in Sect. 3.2. It can be applied to different hardware systems, benefits from the component-based approach for its extensibility and is scalable across multiple machines. Experiments showed a negligible run-time overhead of the framework. We omitted figures here, but the fact that we deploy Fawkes on very different platforms such as our wheeled soccer robots or the Nao, which in particular has restricted computing facilities, shows that the run-time overhead is feasible. The project's website can be found at `http://www.fawkesrobotics.org`.

For our future work, we need to get even more experience with applying Fawkes to different platforms, acquiring more scalability and run-time overhead results, and with offering more sensor and actuator plugins for common robot hardware, in order to serve a broader user community. Other areas of future work are to interface with ROS, as the Robot Operating System is widely used by number of groups. Finally, we want to offer plugins for high-level reasoning. In particular, we are working on a lightweight implementation of Golog [17] and an interface between Golog and Fawkes.

## Acknowledgments

A. Ferrein is currently a Feodor Lynen fellow supported by a grant of the Alexander von Humboldt Foundation. T. Niemueller was partly supported by the German National Science Foundation (DFG). We thank the anonymous reviewers for their helpful comments.

## References

1. Mcilroy, M.D.: 'mass produced' software components. Software Engineering: Report On a Conference Sponsored by the NATO Science Committee (1968) 138–155
2. Szyperski, C.: Component Software – Beyond Object-oriented Programming. Addison Wesley (2002)
3. Brugali, D., Brooks, A., Cowley, A., Côté, C., Domínguez-Brito, A.C., Létourneau, D., Michaud, F., Schlegel, C.: Trends in robot software domain engineering. In Brugali, D., ed.: Software Engineering for Experimental Robotics. Volume 30 of Springer Tracts in Advanced Robotics. Springer-Verlag (2007) 135–142
4. Collins-Cope, M.: Component based development and advanced OO design. Technical report, Ratio Group Ltd. (2001) http://www.markcollinscope.info/W7.html.
5. Chaimowicz, L., Cowley, A., Sabella, V., Taylor, C.: Roci: a distributed framework for multi-robot perception and control. In: Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems. (2003) 266–271
6. Dominguez-Brito, A., Hernandez-Sosa, D., Isern-Gonzalez, J., Cabrera-Gamez, J.: Component software in robotics. In: Proceedings of the 2004 2nd International IEEE Conference on Intelligent Systems. Volume 2., IEEE Press (2004) 560 – 565
7. Côté, C., Brosseau, Y., Létourneau, D., Raïevsky, C., Michaud, F.: Robotic software integration using marie. International Journal of Advanced Robotic Systems **3**(1) (March 2006)
8. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Orebäck, A.: Towards component-based robotics. In: Proc. IROS-05, IEEE Press (2005) 163–168
9. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A.: Orca: a component model and repository. In Brugali, D., ed.: Software Engineering for Experimental Robotic. Springer Verlag (2007) 231–251
10. Niemueller, T., Ferrein, A., Lakemeyer, G.: A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao. In: RoboCup XIII, Springer (2009)
11. Soetens, P.: A Software Framework for Real-Time and Distributed Robot and Machine Control. PhD thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium (May 2006)
12. Henning, M.: The rise and fall of corba. Queue **4**(5) (2006) 28–34
13. Quigley, M., Conley, K., Gerkey, B., Faust, J., Leibs, T.B.F.J., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: Proc. of the Open-Source Software workshop at the International Conference on Robotics and Automation. (2009)
14. Mowbray, T.J.: Architecture in the large. Object Mag. **7**(10) (1997) 24–26
15. Riehle, D.: Framework Design – A Role Modeling Approach. PhD thesis, ETH Zürich (2000)
16. Orebäck, A., Christensen, H.: Evaluation of architectures for mobile robotics. Autonomous Robots **13**(1) (2003) 33–49
17. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A Logic Programming Language for Dynamic Domains. J. of Logic Programming **31** (1997)