Managing extensibility and maintainability of industrial robotics software

Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Hoffmann, Alwin, Andreas Angerer, Andreas Schierl, Michael Vistein, and Wolfgang Reif. 2013. "Managing extensibility and maintainability of industrial robotics software." In *2013 16th International Conference on Advanced Robotics (ICAR), 25-29 Nov. 2013, Montevideo, Uruguay*. Piscataway, NJ: IEEE. https://doi.org/10.1109/icar.2013.6766561.

DOI: 10.1109/ICAR.2013.6766561

Managing Extensibility and Maintainability of Industrial Robotics Software

Alwin Hoffmann, Andreas Angerer, Andreas Schierl, Michael Vistein and Wolfgang Reif
Institute for Software and Systems Engineering
University of Augsburg
D-86135 Augsburg, Germany
E-mail: hoffmann@informatik.uni-augsburg.de

***Abstract*—As neither the set of robotics devices nor the operations they can execute is fixed, a software framework for robotics should be extensible. Moreover, as the environment the robots work in changes, the application controlling them must be easily adaptable to changing requirements. When this can be achieved at run-time, it leads to a continuous evolution of robotics software. This paper presents an object-oriented software framework, the Java-based Robotics API, that facilitates extensibility with code reuse. By integrating the framework into the dynamic module system OSGi, it is possible to continuously evolve a robotics application (including its real-time capable parts).**

I. Introduction

Robots in industrial automation systems are usually tightly integrated in a production environment where they collaborate with other robots and all kinds of machinery. In some areas, e.g. the automotive industry, production systems are set up at most once in every generation of products and then perform the same task thousands of times. While this may (yet) be acceptable in domains like the automotive industry with large batch size production, in many other areas industrial systems are required that can be re-used for different tasks and products. This re-use poses challenges to the robotics hardware (e.g. additional sensors, flexible grippers) as well as to the software (e.g. new robot task or motions) operating them. Hence, the software should be adaptable to changing requirements, modified system components and new hardware platforms. Over time, it must evolve together with the automation system it controls (cf. Koziolek et al. [1]).

In this work, we focus on investigating how an industrial-strength robotics framework must be designed to be easily extensible and moreover to be maintainable – both in a static way (at compile time) and dynamically during run-time. Such a framework can be used to develop evolving automation software by taking advantage of the provided static and dynamic flexibility. The main contributions are to show (1) how extensibility with high code re-use can be achieved across different architectural layers (i.e. the high-level programming interface, the real-time motion control and the low-level device control), (2) how the dependencies between these different layers can be maintained, and (3) how robotic software can be updated or extended during run-time on each of these layers.

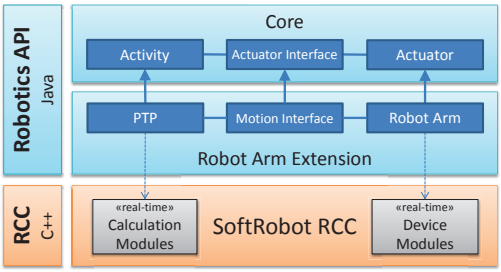

Figure 1. The multi-layered software architecture for industrial robotics developed in the joint research project *SoftRobot*.

The presented framework is based on the software architecture (cf. Fig. 1) developed in the joint research project *SoftRobot* with KUKA Laboratories. The framework is geared towards realizing complex, sensor-guided manipulation tasks of single robots or small teams of robots. The main idea [2] is that robotics software is developed against an application programming interface (API) – the Java-based *Robotics API* [3] – and robot operations are executed with hard real-time guarantees on the underlying *Robot Control Core (RCC)*. Thus, the RCC is responsible for controlling hardware devices and must be implemented on a real-time operating system. The reference implementation – the *SoftRobot RCC* – is implemented in C++ (using libraries from Orocos [4]) and is targeting Linux Xenomai for robot control as well as Windows for testing and simulation purposes.

The Robotics API consists of a core that defines a basic model of controllable devices and combinable operations that these devices can execute. The actual execution is performed on the RCC using real-time capable calculation and device modules. To describe such combinable operations, the RCC is interfaced through a flexible data-flow language [5] by the Robotics API that describes which calculation and device modules must be used and how they must be combined. Such a combination of these real-time capable modules is executed by the RCC with hard real-time guarantees. This integration of flexible high-level application programming and real-time motion execution makes the framework particularly suitable for industrial robotics and distinguishes it from wide-spread component systems such as ROS [6] which have a different (non real-time) focus.

This work partly presents results of the research project *SoftRobot* which was funded by the European Union and the Bavarian government. The project was carried out together with KUKA Laboratories GmbH and MRK-Systeme GmbH and was kindly supported by VDI/VDE-IT GmbH.

978-1-4799-2722-7/13/$31.00 © 2013 IEEE

The core of the Robotics API provides a plug-in mechanism for adding new kinds of devices and operations by appropriate extensions (e.g. for robot arms). By integrating the framework into the dynamic Java-based module system OSGi [7], a fine-grained management of extensions and their dependencies is possible. From our point of view, this is very important for the managed evolution of automation software, because it allows for a controlled integration of new system components and functions to meet changing requirements throughout the life-cycle of the system. Furthermore, such a module system can support dynamic adaptation of the module composition forming the automation software. For example, unloading modules in case of maintenance of hardware components, replacing modules in favor of new, function-enriched versions or adding completely new modules at run-time for scaling up the automation system is possible.

The first part of this paper – Sect. II – presents the principles that were applied to achieve maximum flexibility and extensibility while still maintaining real-time capability. The second part – Sects. III and IV – presents the standardized module system OSGi and our approach to use it for creating maintainable software for automation systems. Subsequently, Sect. V explains how this can lead towards a continuous evolution of robotics software. In Sect. VI, the presented approach is compared to related work. Sect. VII concludes the paper and gives an outlook.

II. Design for Extension: Reuse & Flexibility

The Robotics API is an object-oriented framework for robot programming. An important aspect of object-oriented design is to model software objects that correspond to real-world objects and concepts [8]. The concrete approach of decomposing domain concepts to software objects depends on many factors (e.g. granularity, flexibility, reusability). We found that devices (e.g. robots or sensors) as well as activities (e.g. motions) are well suited to be modeled as software objects in robotics [3]. Separating those two concepts can increase reusability, as neither the set of devices nor the capabilities they can have is fixed.

According to Gamma et al. [9], there are two common techniques in object-oriented systems for reusing functionality. Using *class inheritance*, the implementation of one class can be derived from another (super) class and can be extended on demand (also known as subclassing). This is referred to as *white-box reuse* and is defined statically at compile-time. *Object composition* is a technique where single objects are combined to obtain a new (and more complex) object. This object relies on the functionality of its inner objects and delegates operations to them. This is referred to as *black-box reuse* and can be defined and changed at run-time.

Fig. 2 shows an excerpt from the Robotics API's class hierarchy[1]. It shows the implementation of a KUKA Light-Weight Robot (LWR) based on a generic robot arm. In the following sections, we will investigate the different parts and explain which principles have been applied to achieve both reusability and flexibility.

[1]For purposes of clarity, the distinction between Java interfaces and their (abstract) implementation – a principle of reusable object-oriented software which exists e.g. for actuators and robot arms – was omitted.

A. New devices by inheritance

New robotics devices can be added by subclassing an appropriate abstract super implementation. In Fig. 2, the most general robotics device is an `Actuator` which can be subclassed. Among other attributes and operations, it defines e.g. an abstract method for retrieving the actuator's operation state. Located in another package, the (abstract) class `RobotArm` defines an interface for generic articulated arms and provides an abstract implementation. For instance, it defines that an arm has a base and a flange frame and consists of joints and links. For implementing a concrete robot arm such as the LWR in the example, the `RobotArm` class must be extended and abstract methods have to be implemented (e.g. for creating the specific joints and links). The `Lwr` class is such an specialized robot arm defining additional attributes and methods only common to this type of robot (e.g. a force-torque sensor).

In this case, inheritance facilitates code reuse: the super class already defines the skeleton – i.e. the attributes and methods common to this device type. By using the Template Method Pattern [9], every concrete device is able to (re)define specific implementation details. Apart from the LWR, various other robot types (e.g. KUKA youBot, Universal Robot UR5, Staubli TX90) have been implemented with significant code reuse. For example, the Staubli TX90 robot arm implementation consists of only a few methods defining its links, joints and machine parameters. Apart from robot arms, various device types such as mobile platforms, I/Os, or grippers have been implemented using the same technique. As the device hierarchy is static and can be defined at compile-time, inheritance is – with respect to flexibility and code reuse – an appropriate technique here.

B. Adding & exposing new robot capabilities

While the structure of robotic devices is fixed, their capabilities, e.g. the actions or motions they can execute, are not. Thus, actuator implementations in the Robotics API, like `RobotArm`, do not provide a set of methods for executing certain actions. Instead, an `Actuator` is modeled as a composition of `ActuatorInterfaces` which define semantically related operations (e.g. motions). Concrete implementations are provided at run-time by suitable actuators (e.g. a `RobotArms` provides an implementation of `MotionInterface`). Here, composition was chosen over inheritance to be flexible but also to enable re-use by sharing implementations among similar actuators (e.g. among standard six-axis robot arms of different manufacturers).

To combine robot activities and to monitor their execution progress, methods of `ActuatorInterfaces` do not directly execute operations, but return objects of type `Activity` [3]. This class resembles a command object as suggested by the Command Pattern [9]. As our approach is using a standard Java VM, an `Activity` is transformed into a data-flow description of calculation and device modules (cf. Fig. 1) that can be executed by the RCC with real-time timing guarantees. On the Java side, the `Activity` is only a proxy monitoring the execution progress (cf. [3]). New activities can be developed by subclassing the `Activity` class or an appropriate subclass (e.g. `MotionActivity`). Moreover, activities can be composed [9] using typical combinations such as sequential or parallel execution. Thus existing activities can be re-used as part

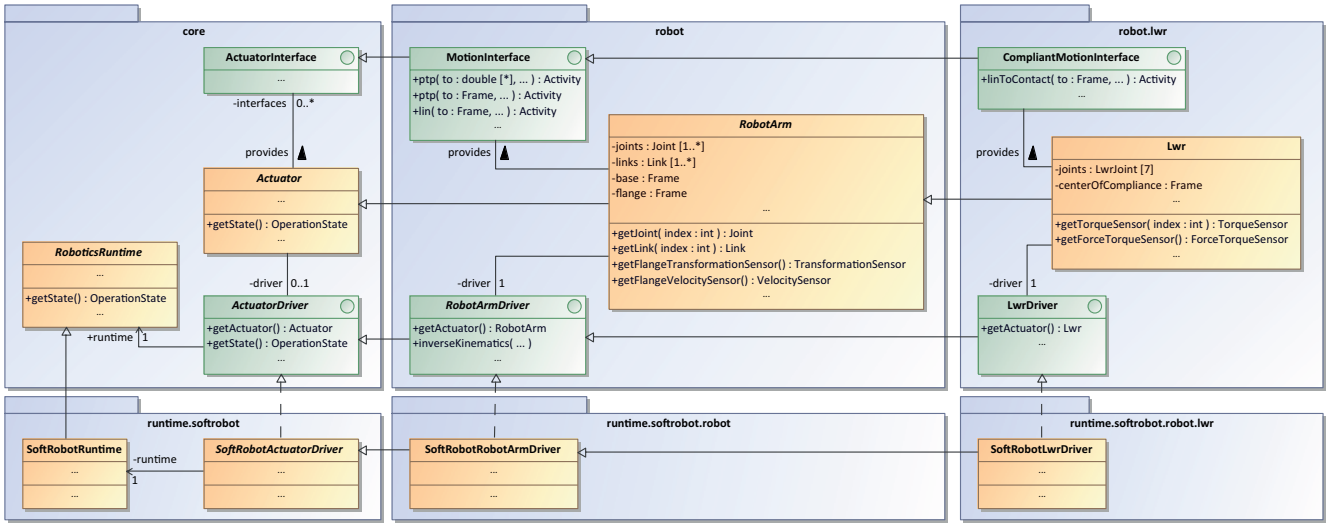

Figure 2. UML class diagram with an excerpt from the Robotics API's object class hierarchy focusing on the modeling and implementation of robot arms.

of more complex operations. The composition mechanisms are designed such that they preserve real-time guarantees for the execution of composed activities.

C. Delegating device control & communication

`Actuators` and their subclasses (e.g. the `RobotArm`) define the interfaces and the structure of different device types and concrete devices. Any operation involving communication to the device hardware (e.g. for retrieving current joint values or for executing motions) is delegated to a so called `ActuatorDriver` (cf. Fig. 2) which are responsible for the communication with the real device. As such, their type is specific to the device type (e.g. the `RobotArmDriver`) and their implementation is specific to the way of communication with the device hardware. Thus, by applying the principle of delegation, loose coupling between the application programming and low-level device communication is achieved.

In the Robotics API, the implementation of drivers is specific to the underlying RCC which is represented by a particular implementation of the class `RoboticsRuntime`. For instance, the *SoftRobot RCC* reference implementation (cf. Sect. I) is represented by the `SoftRobotRuntime` class. Consequently, there are driver implementations specifically for this runtime (e.g. the `SoftRobotLwrDriver` for controlling LWRs via the SoftRobot RCC). It is the responsibility of those concrete `ActuatorDrivers` to map operations modeled by `Activities` to corresponding real-time capable calculation and device modules of the underlying RCC (cf. [10]).

By separating activities from actuator definitions as explained in the previous section, the mapping from high-level definition of operations to their real-time capable equivalents in the RCC is for the most part independent of concrete actuators. For example, the combination of real-time RCC modules to use for interpolating a linear motion can be done for many types of robot arms in a generic way. The `SoftRobotRobotArmDriver` serves as a generic driver which can be used to execute basic motions. Implementing support for a concrete robot becomes very simple. For the aforementioned Staubli TX90 robot, the only new functionality that had to be implemented was a real-time capable RCC module for connecting to the robot using Staubli's uniVAL interface. Already existing functionality, in particular the implementation of different motion types, was reused from the robot arm driver.

Apart from using drivers, an actuator's behavior can be implemented using other (driver-based) actuators. When applying composition, the composed actuator must derive its internal state completely from its inner objects and its behavior must be completely mapped onto a composition of their behavior (e.g. by a composition of activities). Typical usages for composed actuators are robot tools which are interfaced by digital and analog I/Os (e.g. parallel grippers from Schunk's MEG series). However, composition can be also used to combine single actuators to a more complex actuator (e.g. combining a robot arm and a mobile platform) or to implement derived sensors (e.g. combining two laser range sensors).

III. OSGi at a Glance

The OSGi framework is a dynamic module system and service platform for Java. Since 1999, its specification [7] has been developed by the members of the OSGi Alliance (e.g. IBM, Oracle, Samsung, Siemens) in an open process and is publicly available. The OSGi framework is used e.g. in application servers, automotive systems or the Eclipse IDE.

An application based on OSGi is composed of many different (reusable) components called *bundles*. As in plain Java, bundles are packaged and deployed as JAR files. However, bundles are self-describing by using a *manifest* file to declare their public API (i.e. exported packages) and thus to hide internal implementation. In addition, the manifest defines runtime dependencies on other packages or bundles (of a specific version). OSGi applications are dynamic in the sense that the set of bundles can change at run-time. Bundles have a life-cycle and can be installed, updated, or even uninstalled at any time. Hence, there is no *main* program: bundles are activated by the framework and, for example, may start threads or collaborate with other bundles.

For collaboration among bundles, OSGi implements a service model and introduces a *service registry*. Using this

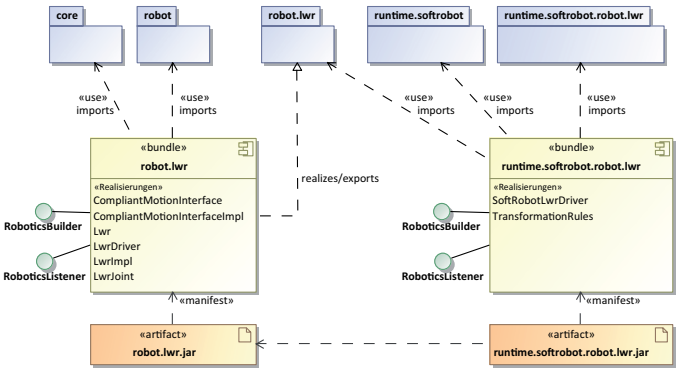

Figure 3. UML component diagram showing bundles which realize two exemplary packages from Fig. 2.

registry, a bundle can register or retrieve a service, and it can listen for services to appear or disappear. Hence, different bundles can register the same type of service, and different bundles can use the same service. Similar to bundles, services are also dynamic, i.e. a bundle can decide to withdraw its services from the registry. Other bundles using a withdrawn service are informed about this and must then ensure that they drop any references to the service object. To facilitate this, OSGi offers techniques such as Declarative Services [11]. Based on the service model, OSGi additionally provides an universal event mechanism that allows to publish and consume events by giving a topic

IV. Maintainability at Run-time

Sommerville [12] defines maintainability as the ability of a software system to economically cope with new emerging requirements. ISO 9126 [13] describes maintainability as the ability of the software to be modified with relatively little effort. Such a modification can be the correction of bugs, an improvement, or the adaptation to a changed environment. Due to its explicit definition of public API and its dynamic module system, OSGi is well suited to create maintainable software [14]. Sect. IV-A shows how these abilities can be exploited for robotics. In Sect. IV-B, additional challenges for maintainable software in robotics (i.e. real-time device control) are described and solutions are sketched.

A. Deployment: Packaging as bundles and using services

Each package from Fig. 2 is deployed into a separate bundle describing its public API and its run-time dependencies to other packages/bundles. The diagram in Fig. 3 show this for two exemplary packages. On the left hand side, the `robot.lwr` bundle is shown. It is an API bundle which exports the corresponding package and, thus, realizes its interfaces and classes. In order to do this, it relies on two imported packages: `core` and `robot` with regard to a specific version. This bundle does not export every implemented class but rather provides these implementations by using services as explained later. On the right hand side of Fig. 3, the runtime-specific bundle `runtime.softrobot.robot.lwr` is shown. It realizes a runtime-specific driver for the KUKA LWR and defines required real-time (calculation) modules (e.g. for compliant motions). While it does not export any public API, it imports other packages such as `robot.lwr` which was exported by the previously described bundle.

As mentioned before in Sect. III, services should be used for the collaboration among bundles, i.e. services should be shared and not concrete implementations. For example, the `runtime.softrobot.robot.lwr` bundle offers a *RoboticsBuilder* service to create instances of the internal `SoftRobotLwrDriver` implementation (cf. the Builder Pattern [9]). This approach ensures that application programmers neither create instances of nor program against internal implementation classes. As builders are registered as services, infrastructure components can retrieve them using the OSGi service registry and use them to instantiate the required actuators and drivers. The process of creating, configuring and publishing actuators and drivers is application-specific. For example, the set of required actuators and drivers can be defined in a configuration file, and actuators can be either published as services or stored in a registry.

Before publishing an actuator, it is important to inform all listeners implementing the *RoboticsListener* interface. Both bundles from Fig. 3 implement this interface and offer it as a service using the Whiteboard Pattern [15]. By doing so, bundles are informed by the infrastructure component whenever a new (properly configured) actuator, runtime, and driver is added or removed. Moreover, new listeners are informed about existing actuators, runtimes, and drivers. Hence, listeners can be used to add and remove actuator interface implementations to suitable actuators (e.g. the `CompliantMotionInterfaceImpl` is added to every built `Lwr` to provide the `CompliantMotionInterface`). Runtime-specific bundles can use this interface to register transformation rules for drivers that define which real-time modules are expected inside the RCC and how high-level operations (i.e. `Activities`) are mapped to these real-time modules (cf. Sect. II-C).

B. Dependencies: Java bundles and real-time modules

Deploying an OSGi-based application means to deploy a set of bundles that constitute this application. In the case of a Robotics API application, API bundles, runtime-specific bundles as well as application-specific bundles need to be deployed. Fig. 4 shows the dependencies between these kinds of bundles at run-time. The dependencies between packages (cf. Fig. 3) are resolved and mapped to dependencies between bundles. As version ranges can be specified for dependencies, OSGi takes care on resolving this adequately and allows multiple versions of the same class definition inside a single Java VM. If a bundle's dependencies are resolved, it is installed and its services are registered.

The API bundles have dependencies on each other according to the class hierarchy from Fig. 2, e.g. the `robot` bundle depends on the `core` bundle and imports interfaces and classes from there. The application (represented by a single `application` bundle) only depends on API bundles, i.e. only interfaces and classes of API bundles are used. This is enforced as runtime-specific bundles should not export any implementation classes. By doing so, the application is completely independent from the used runtime and RCC.

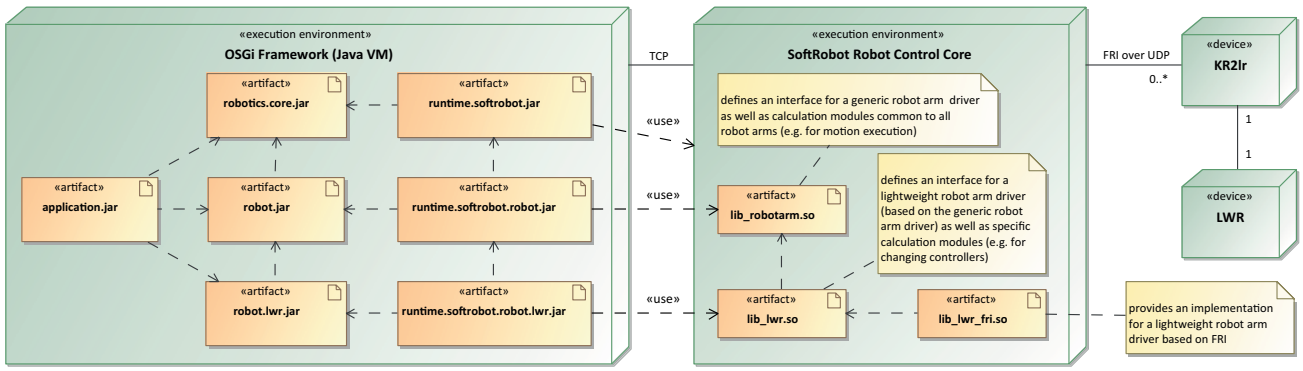

Figure 4. UML deployment diagram showing the dependencies between bundles. Similar to runtime-specific bundles, the application bundle depends on API bundles. Besides these dependencies inside the OSGi Framework, there are additional dependencies between runtime-specific bundles and (real-time) modules (packaged as shared object libraries) inside the RCC. Transitive dependencies (e.g. the `robot.lwr` bundle also depends on the `core` bundle) are omitted here.

Runtime-specific bundles have dependencies on API bundles as they are using classes and interfaces from there (e.g. the `RobotArmDriver` interface). Moreover, they also have dependencies on each other according to the object class hierarchy. The OSGi framework is able to resolve these dependencies automatically. However, every runtime-specific bundle has additional dependencies (in the form of usages) to the RCC and to modules inside the RCC as shown in Fig. 4. For example, the `SoftRobotRuntime` class (located in the `runtime.softrobot` bundle) implements the network communication to the RCC reference implementation and, thus, relies on a certain communication protocol. Other runtime-specific bundles rely on a set of certain real-time calculation modules present at the RCC. For instance, the `runtime.softrobot.robot` bundle assumes that a specific set of real-time RCC modules is available for interpolating a linear motion (cf. Sect. II-C). This dependency cannot be handled automatically as the real-time modules (C++) are part of the RCC and, thus, located outside the Java-based OSGi framework (cf. Fig. 4).

In order to develop maintainable robotics software, a matching between bundles (Java) and real-time modules (C++) has to be performed at run-time. To achieve this, every runtime implementation (e.g. the `SoftRobotRuntime`) can validate whether its version matches the RCC while connecting. Moreover, it has to provide means to validate whether runtime-specific bundles match to installed real-time modules on the RCC which is feasible because there is a one-to-one relationship between those bundles and modules (cf. Fig. 4). For this validation, we employ fragments, a special form of OSGi bundles, which are always attached to another bundle, the fragment host. A fragment is treated as part of the host bundle and its contents are made available to this bundle. Fragments are normally used to extend bundles with resources or platform-specific libraries. Hence, for each supported RCC operating system (e.g. Linux Xenomai, Windows) a shared library (C++) containing the required real-time modules is packaged inside a dedicated fragment and can be deployed from the OSGi framework (Java) to the RCC (C++). For example, there are different fragments for the `runtime.softrobot.robot` bundle. Each fragment contains the `lib_robotarm` shared library for a particular operating systems (e.g. `lib_robotarm.so` for Linux Xenomai, see Fig. 4). This deployment mechanism allows both for validating a particular shared library (using check sums) and for remotely installing the shared library and the contained real-time modules onto the RCC at run-time.

V. Towards Continuous Evolution

Because OSGi manages all dependencies between bundles, bundles and, thus, actuators or drivers can be updated during run-time. To update a bundle, it first has to be uninstalled. Hence, its services are unregistered, which is recognized by an infrastructure component. The infrastructure component will then invalidate and remove all actuators and drivers provided by the uninstalled bundle. Accordingly, it removes every `ActuatorInterface` implementation provided by the bundle. This can lead to a degraded mode where the application is not able to work properly, because e.g. a required robot (object) is not present anymore. However, the application must support this evolution process properly and the point in time when the system is updated should be chosen wisely in order to be able to resume working afterward.

To replace the bundle, a new version must be installed and started. In doing so, it registers its services again. Bundles, that have dependencies to this replaced bundle or to packages exported by it, have to be refreshed. By refreshing dependent bundles, these bundles are getting stopped and started again. If a dependent bundle is not compatible with the new bundle version, it will not be started by the framework. Hence, OSGi takes care of restoring a consistent composition of bundles. As new services are getting registered, the infrastructure component will try to restore the previously removed actuators and drivers, however using the newly installed bundle. If this can be accomplished successfully, the application can resume working properly.

As real-time modules for the RCC are also packaged in runtime-specific bundles (respectively in attached fragments) and provided by OSGi services, this process can be used to install new or update existing real-time modules (by transferring a new shared library to the RCC). This counts for calculation modules, which may be required for the execution of new `Activities`, as well as for device modules, which are responsible for establishing the low-level communication with hardware devices (cf. Fig. 1). In order to install a real-time module, the target platform (e.g. the operating system) of the RCC is determined and the appropriate fragment, which

includes information about the target platform, is resolved. Subsequently, the shared library is transferred from the fragment to the RCC and the contained real-time modules are installed there. When replacing a runtime-specific bundle (and its fragments) the provided real-time modules are uninstalled from the RCC and replaced by new modules provided by the updated bundles (and its fragments respectively).

Robotics software development can take advantage of the mechanisms a dynamic module system such as OSGi provides. To a certain extent, it allows for a continuous evolution of the system (including the real-time control of devices) at runtime. Moreover, the dynamic nature of OSGi can be used to develop highly flexible and sustainable automation systems. To achieve this, automation systems must be modeled using the service-oriented paradigm, i.e. every automation component is modeled as a service and, as such, is exchangeable. This allows for updating or replacing parts of the automation system at run-time. If the process- or workpiece-specific parts are also modeled as services, the automation system is to certain extent adaptable to changing requirements. First results are very promising.

VI. Related Work

For managing the complexity of large-scale application scenarios, Hägele et al. [16] identified the *decomposition of systems into components* and the *composability of subsystems* as core requirements. It is addressed by most component-based systems like Player [17], ROS [6], YARP [18] or ORCA [19]. Because in such systems the application is completely composed of loosely coupled components, they are extensible by design. However, many of the available component frameworks do not particularly support hard real-time execution of robot tasks, which makes it harder to achieve the precision required in industrial applications. The Orocos [4] framework, which is used for our RCC reference implementation, supports real-time execution and is extensible with reusable components. However, we added an extensible and easy-to-use API layer above real-time robot control which makes in necessary to achieve reusability and to maintain dependencies over multiple architectural layers.

According to Brugali et al. [20], reusability in component-based software engineering is achieved by separating the component specification (i.e. its *provided* and *required* interfaces) and its implementations which may vary in functional or non-functional characteristics. As components are used as *black boxes*, they can be reused in various applications. To facilitate *black box reusability*, Brugali et al. [21] propose a reuse-oriented development process. Mallet et al. [22] make suggestions on how to design robot components. The techniques proposed by Brugali et al. [20] for reducing the implementation effort for a component specification are the same object-oriented paradigms which have been successfully applied in our approach (e.g. inheritance, delegation). The component-based CLARAty framework [23] e.g. also uses class inheritance to achieve extensibility and code reusability. Other approaches that use object-oriented concepts for reusability are MARS [24] and RIPE [25]. Similar to CLARAty, they neither regard maintainability of robotics software nor allow for evolving the system during run-time. Due to its automatic

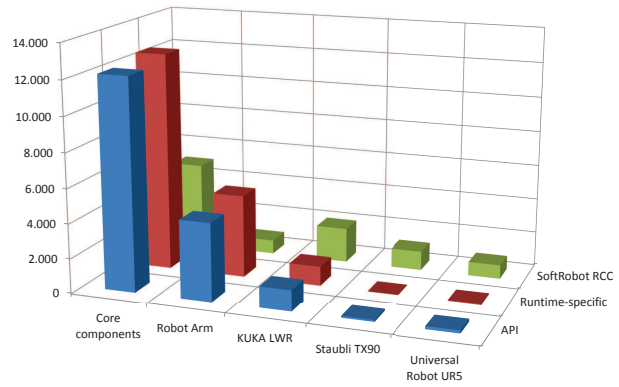

Figure 5. Lines of code distribution across the artifacts shown in Fig. 4.

management of dependencies, *white box reusability* is supported very well by OSGi which is exploited in our framework.

Robot frameworks often act as a middleware by organizing the communication between components. For example, ROS uses service calls and message-passing (topic-based publish/subscribe) for communication between different processes which are called nodes. By only exchanging messages, components are loosely coupled and only rely on a common specification of message types. Thus, it is possible to stop a running node and replace it with an updated version. However, other nodes can hardly recognize that the node is absent for some time. By replacing a node, the application-specific device interface and the low-level communication with the device can also be updated. As frameworks like ROS usually do not support real-time communication, the application-specific device interface cannot rely on real-time guarantees (e.g. it must use velocity control instead of cyclic position control) and the evolution of any required real-time code must be done manually while deploying or starting the new node.

Georgas and Taylor [26] have examined different architectural styles found in robotics and have elaborated if and how they support run-time evolution. They come to the conclusion that "a lack of explicit architectural layering and clearly encapsulated behaviours" [26] are the most important shortcomings that prevent evolving robotics systems at run-time. However, both shortcomings are addressed in the approach presented in this paper which leads to the desired software evolution at run-time. Luo et al. [27] also present an OSGi-based approach for software evolution. They are using behavior networks [28] where each available behavior is represented as a bundle. New behavioral bundles and new behavior networks can be deployed at run-time using the dynamic character of OSGi. However, they do not address reusability and real-time execution of robot operations.

VII. Conclusion

This paper shows that an easily extensible robotics framework can be developed that (1) supports robot programming including the real-time execution of robot operations and (2) facilitates the reusability of source code. To illustrate reusability, Fig. 5 shows the lines of code of API bundles, runtime-specific bundles and real-time modules that are necessary to support KUKA's LWR, Staubli's TX90 and Universal Robot's UR5. It shows that most of the code was implemented for basic

Robotics API concepts (e.g. `Actuators` and `Activities`) as well as the runtime-specific mapping of these concepts to real-time modules and the communication to the RCC. The generic Robot Arm extension includes the path planning for different motions and contains a considerable amount of transformation rules to map these motions to real-time modules. The Staubli extension as well as the UR5 extension only requires minimal Java code to e.g. define the link/joint structure and default parameters, while other functionality is re-used from the generic Robot Arm extension. The LWR requires some additional Java code which is specific for its torque sensors, controller modes and force-guarded motions. In the SoftRobot RCC, the most part of code is required for the core components, as well as for implementing the low-level communication with concrete robot arms, while the generic Robot Arm extension is relatively slim.

However, the right deployment plays a significant role for the maintainability of (robotics) software. OSGi allows for explicitly defining a bundle's public API and its dependencies to other packages and bundles of a specific version. Special attention should be paid to not publishing particular implementations, but using services. The dynamic service model not only allows to add new devices, driver implementations and robot capabilities, but also to update existing devices and driver implementations as well as real-time modules. Hence, it enables and facilitates not only the continuous evolution of robot-based automation systems but also of service robots which is getting more and more important in the future [29].

As applications only have dependencies to API bundles, they are completely independent from the underlying RCC. Hence, runtime-specific bundles do not need to be bound at compile time. Instead, it is possible to provision these bundles for the first time when the application is started. Experiments using an OSGi-based marketplace showed that it is possible to download and install these bundles at run-time. Because OSGi offers mechanisms for network communication with other systems, it can be considered as a service-oriented middleware for Java. Remote Services [11] allow for adapting the service model and event mechanism to distributed OSGi systems, web services, and for communication with other processes. This can be used to adapt the Robotics API and its abilities for evolving at run-time to other (robotics) middlewares. While support for the Device Profile for Web Services [30] has already been implemented, further adaptations are future work.

REFERENCES

[1] H. Koziolek, R. Weiss, Z. Durdik, J. Stammel, and K. Krogmann, "Towards software sustainability guidelines for long-living industrial systems," in *Proc. 3rd Workshop of GI Working Group 'Long-living Software Systems (L2S2)': Design for Future 2011*, ser. LNI, 2011.

[2] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif, "Hiding real-time: A new approach for the software development of industrial robots," in *Proc. 2009 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, St. Louis, MO, USA*, 2009, pp. 2108–2113.

[3] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics API: Object-Oriented Software Development for Industrial Robots," *J. of Software Engineering for Robotics*, vol. 4, no. 1, pp. 1–22, 2013.

[4] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proc. 2001 IEEE Intl. Conf. on Robot. & Autom., Seoul, Korea*, 2001.

[5] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. IEEE Intl. Conf. on Autom. and Logistics, Hong Kong*, 2010.

[6] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *IROS Workshop on Open Source Software*, 2009.

[7] "Osgi core release 5," OSGi Alliance, Mar. 2012. [Online]. Available: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf

[8] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall, 2004.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison Wesley, 1994.

[10] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif, "From robot commands to real-time robot control: Transforming high-level robot commands into real-time dataflow graphs," in *Proc. 9th Intl. Conf. on Inform. in Control, Autom. & Robot., Rome, Italy*, 2012.

[11] "Osgi enterprise release 5," OSGi Alliance, Mar. 2012. [Online]. Available: http://www.osgi.org/download/r5/osgi.enterprise-5.0.0.pdf

[12] I. Sommerville, *Software Engineering*, 8th ed. Addison Wesley, 2007.

[13] "Iso/iec 9126 software engineering – product quality," ISO.

[14] J. McAffer, P. VanderLei, and S. Archer, *OSGi and Equinox - Creating Highly Modular Java Systems*. Addison Wesley, 2010.

[15] P. Kriens and B. Hargrave, "Listeners considered harmful: The whiteboard pattern," OSGi Alliance, Whitepaper, 2004.

[16] M. Hägele, K. Nilsson, and J. N. Pires, "Industrial robotics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 42, pp. 963–986.

[17] T. Collett, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proc. 2005 Australasian Conf. on Robotics and Automation*, Sydney, Australia, Dec. 2005.

[18] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robot. Auton. Syst.*, vol. 56, no. 1, pp. 29–45, Jan. 2008.

[19] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for Robotics," in *Workshop on Robotic Standardization. IEEE/RSJ Intl. Conf. on Intell. Robots and Systems*, Beijing, China, Oct. 2006.

[20] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robot. & Autom. Mag.*, vol. 16, no. 4, pp. 84–96, 2009.

[21] D. Brugali, L. Gherardi, A. Biziak, A. Luzzana, and A. Zakharov, "A reuse-oriented development process for component-based robotic systems," in *Proc. Simulation, Modeling, and Programming for Autonomous Robots*, ser. LNCS. Springer, 2012, vol. 7628.

[22] A. Mallet, F. Kanehiro, S. Fleury, and M. Herrb, "Reusable robotics software collection," in *Proc. 2nd Wksh. on Softw. Developm. & Integr. in Robotics. IEEE Intl. Conf. on Robot. & Autom.*, Rome, Italy, 2007.

[23] I. A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, "CLARAty and challenges of developing interoperable robotic software," in *Proc. 2003 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems*, Las Vegas, USA, Oct. 2003, pp. 2428–2435.

[24] G. McKee, J. A. Fryer, and P. Schenker, "Object-oriented concepts for modular robotics systems," in *Proc. 39th Intl. Conf. & Exh. on Techn. of Object-Oriented Lang. & Syst.*, Santa Barbara, USA, 2001.

[25] D. J. Miller and R. C. Lennox, "An object-oriented environment for robot system architectures," *IEEE Control Syst. Mag.*, vol. 11, no. 2, pp. 14–23, 1991.

[26] J. Georgas and R. Taylor, "An architectural style perspective on dynamic robotic architectures," in *Proc. 2nd Wksh. on Softw. Developm. & Integr. in Robotics. IEEE Intl. Conf. on Robot. & Autom.*, Rome, Italy, 2007.

[27] S. Luo, P. Jiang, and J. Zhu, "Software evolution of robot control based on OSGi," in *Proc. IEEE Intl. Conf. on Robot. & Biomimetics*, Shenyang, China, 2004.

[28] P. Maes, "Situated agents can have goals," *Robotics and Autonomous Systems*, vol. 6, no. 1, pp. 49–70, 1990.

[29] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The use of reuse for designing and manufacturing robots," RoSta Consortium, White Paper, 2009.

[30] "Devices profile for web services," OASIS, Jun. 2009. [Online]. Available: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01