Evolving Industrial Software Architectures into a Software Product Line: A Case Study

Heiko Koziolek, Roland Weiss, and Jens Doppelhamer

ABB Corporate Research, Industrial Software Systems,
Wallstadter Str. 59, 68526 Ladenburg, Germany,
{heiko.koziolek | roland.weiss | jens.doppelhamer }@de.abb.com

Abstract. Industrial software applications have high requirements on performance, availability, and maintainability. Additionally, diverse application landscapes of large corporate companies require systematic planning for reuse, which can be fostered by a software product-line approach. Analyses at the software architecture level can help improving the structure of the systems to account for extra-functional requirements and reuse. This paper reports a case study of product-line development for ABB's robotics PC software. We analysed the software architectures of three existing robotics applications and identified their core assets. As a result, we designed a new product-line architecture, which targets at fulfilling various extra-functional requirements. This paper describes experiences and lessons learned during the project.

1 Introduction

The high requirements for extra-functional properties, such as performance, availability, and maintainability, in industrial software applications demand carefully designed software architectures. Industrial software applications control complex machines and have to respond to user requests or other external stimuli within strict time constraints to avoid failures and harm to human beings. They have to exhibit a high availability with very limited down-time to provide maximal benefit for customers. Internally, they should be structured to allow for efficient maintenance and long-term evolution. All these features should be enforced by the underlying software architecture.

Large corporate companies, which serve multiple application domains, have to deal with diverse software application landscapes that complicate fulfilling all extra-functional requirements. In our context, we analyzed the situation for the robotics software at ABB. There are more than 100 software applications in the robotics domain from ABB. These applications have been developed by distributed development teams with limited centralized planning and coordination. This situation has accounted for a high functional overlap in the applications, which has lead to high and unnecessary development and maintenance costs.

A common solution for this problem is the introduction of a software product-line [1], which systematically targets at bundling common assets and building customized applications from reusable software components. Many companies, such as Nokia, Philips, and Bosch have successfully introduced software product lines. While several studies have been reported (e.g. [2–4]), which aim at deriving product-line architecture from existing software application, no cookbook solution can be applied for industrial software applications so far.

In this paper, we report our experiences from 3 year running project at ABB Research reconstructing and evolving the software architectures of three robotics PC applications from ABB. We analyzed the different applications for their shared functionalities and special advantages. We identified core assets and bundled common functionality into reusable software components. We designed new interfaces and ultimately developed a software product-line architecture to systematize reuse among the applications. During the course of the project, we learned several lessons, which could be interesting both for other software architects and researchers.

The contribution of this paper is a case study on architecture evolution and software product-line design in the industrial application domain. The case study includes experiences and findings, which could stimulate further research. We used and assessed different methods from research for the benefits in our domain.

This paper is organized as follows: Section 2 reports on a survey of ABB robotics software, which revealed a significant functional overlap and little reuse. The application domain and the three applications we analyzed are described in more detail in Section 3. Section 4 elaborates on the three phases of our architecture evolution and product-line development project. Section 5 summarizes our lessons learned, and Section 6 surveys related work. Finally, Section 7 concludes the paper and sketches future work.

2 The Challenge: Functional Overlap

A comprehensive survey on ABB's robotics software was conducted in 2006 and motivated our project. The software application landscape within ABB Robotics is diverse and scattered with over 120 applications developed in 8 different countries (mainly Sweden and Norway) by 10 different organizations. The software supports a large number of robot application domains, such as arc welding, spot welding, press automation, painting, sealing, material handling, etc.

The used programming languages include C, C++, C#, VB, and JavaScript. Furthermore, a Pascal-like imperative language called RAPID is used by many applications for implementing robot programming logic for ABB's main robot controller called IRC5. Several applications target the Windows operating system, while other applications run directly on robot controllers using the VxWorks real-time operating system. The code complexity of the applications ranges from small tools with 1 KLOC to large applications with more than 600 KLOC.

The survey analysed 58 ABB robotics applications in detail. Fig. 1 shows a high-level overview of the application landscape. The 58 applications depend on 13 base elements, which provide functionality for example for remote communication and graphical user interfaces. The applications themselves provide different extension interfaces to allow user-specific customization. However, apart from the base elements there is very few reuse among the applications as depicted by the low number of dependencies between the applications in Fig. 1.

Therefore, the survey broke down the functionalities of the applications in detail and categorized them into 30 different functions. Each application developer group was asked what functionality their tool implemented. Fig. 2 shows a condensed view of the results. The left-hand side depicts the number of applications implementing the same function. For example, function 1 was implemented repeatedly in 11 different applications.

It could be argued that the low amount of reuse results from the distinct robot application domains, where software applications are implemented without regard of other application domains. Therefore, the right-hand side of Fig. 2 shows the number of functions, which where implemented multiple times within a single application domain. For example, in domain 1 developers have implemented 161 functions multiple times in different applications.

The low level of reuse among the applications contributes to the high maintenance costs for the applications, which the survey found to be in the range of several million US-dollars per year. As expected, the most complex applications have the highest maintenance costs. However, the survey also identified some outlier applications with unproportionally high maintenance costs despite a small amount of code.

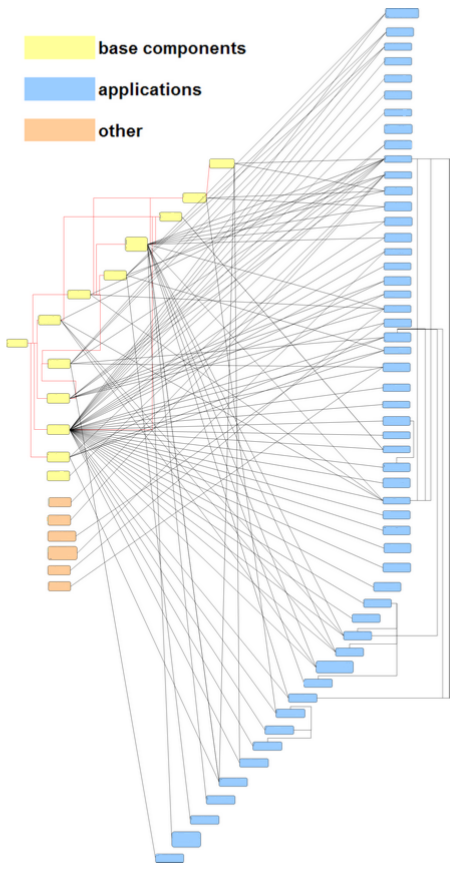


Fig. 1. Application Landscape of ABB's Robotics Software (schematic view, anonymised)

There are several *reasons* for the undesirable functional overlap within ABB Robotics applications. The organisational structure of the software development units has no central unit coordinating and organizing systematic reuse. Several company acquisitions into the corporate body have contributed to the situation. The software is created by a large number of development teams, sometimes

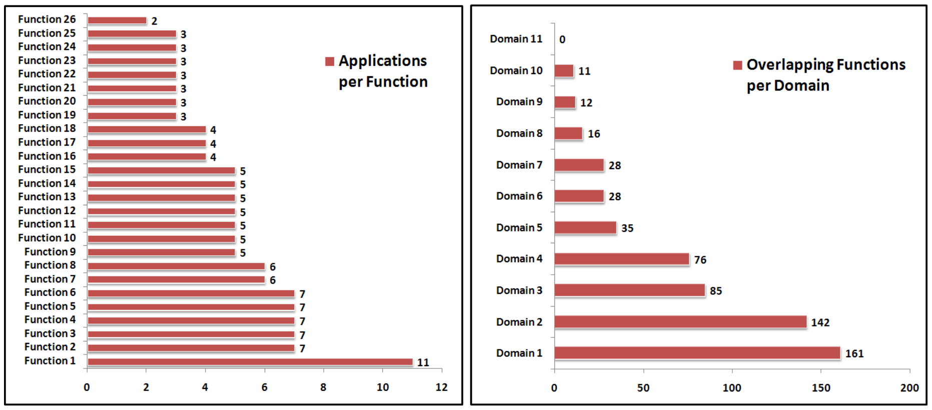


Fig. 2. Functional Overlap in ABB Robotics Software

consisting only of 2-3 developers. The communication among the teams is limited within and across application domains and organizational units.

Because of the large size of the application landscape, no developer can follow which applications are developed in other units and where reuse potential might be high. Some applications are small and tailored for very specific functions and thus exhibit limited reuse potential. Other applications are large, but their software architecture is documented only in a limited way, so that it is not easy to isolate certain functions in the code.

The amount of functional overlap bears high potential for sharing code (i.e., reusable components) among the applications. To decrease maintenance costs and time-to-market, the survey suggests to bundle more common functionality into reusable software components (e.g., COM components or .NET components). More communication among the development units is needed and a central organization for planning systematic reuse would be desirable.

Notice that the survey did not involve all ABB robotics software, therefore the potential for reuse might be even higher. We believe that this situation of functional overlap is not specific to ABB, but common for large corporate companies, which serve different applications domains and rely on distributed development teams. More research should be devoted to documenting, analysing, and redesigning complex application landscapes (cf. [5]).

3 Systems under Study:

With the challenge of functional overlap in mind, we started an architecture redesign project in 2006 focussing on ABB robotics PC applications. Applications running on embedded devices were out of scope for our project. This section first briefly describes the robotics PC application domain (Section 3.1) to let the reader understand the extra-functional requirements for these systems. Then, it sketches the high-level software architectures of three PC applications, which were the basis for our project (Section 3.2), and lists the extra functional requirements (Section 3.3).

3.1 Application Domain

Fig. 3 depicts a typical industrial robot system. It may involve a single robot for a specific task or a whole robot line consisting of several robots arranged in subsequent order.

One or several robot controllers handle movements of the robot arms. This involves path planning and robot axis control. Mustapic et al. [6] have detailed on the open platform architecture of ABB's robot controller. It is an embedded system consisting of ca. 2500 KLOC in C divided into 400-500 classes. The controller kernel provides special support for implementing application specific extensions. The robot controller has an attached teach pendant, a small handheld device for manual control and supervision of a robot. The robot controller and its extensions typically run on a embedded operating system such as VxWorks or Windows CE.

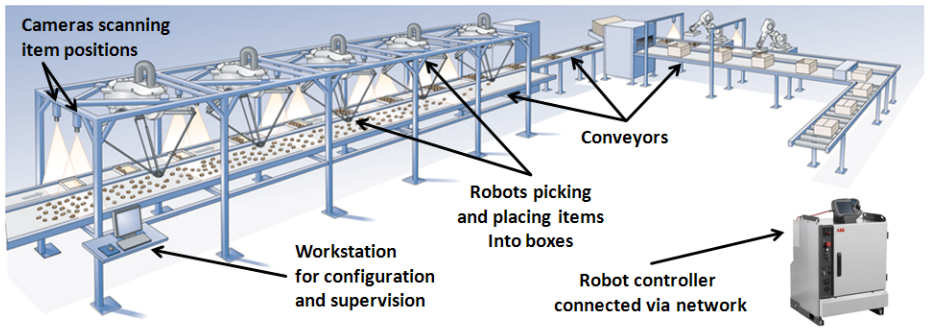

Cameras scanning item positions
Conveyors
Robots picking and placing items Into boxes
Workstation for configuration and supervision
Robot controller connected via network

Fig. 3. Exemplary Industrial Robotics System for Packaging: Overview

The robot system can include a number of external sensors, such as cameras for scanning items to be processed or automatic scales for weighting items. Multiple conveyor belts may feed the robots with items and forward the processed items. In larger robot systems, operators supervise the whole process supported by an arbitrary number of PC workstations, which for example visualize scanned items or allow manipulating robot configurations.

The following coarse functionalities are carried out by PC applications in such a robot system:

- **Engineering:** deals with offline robot programming on an office PC, which is decoupled from robot production. It allows configuring and preparing robot

programs in advance without interfering with robot production. Modern robot engineering tools offer 3D visualizations of robots systems to assist path planning.
- **Simulation:** allows testing the robot installations and programs (i.e., RAPID code) created during engineering. This functionality is similar to program debugging, as it allows to set break points and execute the programs step-by-step. It targets identifying robot collisions and analysing robot reachability.
- **Supervision:** lets operators monitor and control the robot system. This includes consolidating logs from different robot controllers and visualizing the data collected from different robot controllers.
- **Job Control:** manages a job queue and controls the execution of jobs from the queue. A job captures a task the robot system shall execute, for example painting a car or picking and placing a certain amount of goods. Controlling the job queue may involve simply executing RAPID code on the robot controllers, or, in more complex cases, collecting and analysing data from external sensors and distributing item coordinates to different robot controllers.
- **Job Coordination:** coordinates jobs running on multiple job controllers during production. Job coordination for example allows synchronizing different jobs in a robot system, so that subsequent jobs execute with minimal delay, or switching jobs on multiple controllers in a coordinated way (e.g., a new color for painting the next car has been chosen and all involved robots have to adjust accordingly).

Additionally, attached programmable logic controllers (PLC) are used for coordinating the robot system within the context of superordinated systems. For example, information from ERP systems or other production systems can be used to direct robot execution.

3.2 Initial Architectures

We analyzed three different PC applications, each one targeting a specific application domain. These applications were chosen because of their considerable value to ABB business and their perceived similarities. The picking/packing application supports high speed packing of small goods. The painting application for the automotive industry supports colouring cars. The palletizing application supports piling and unpiling goods onto pallets. The three applications have been implemented by different development teams and only exhibit limited reuse among each other.

Fig. 4 shows the high-level software architectures of the three applications in a component-and-connector view. The functionalities described in Section 3.1 have been implemented differently in the the different applications.

The picking/packing application combines engineering, supervision, and job control in a single software tool. In this case, the job control functions involve scanning captured camera images for item positions and distributing the item positions to different robot controllers. The tool is a typical Win32 application with

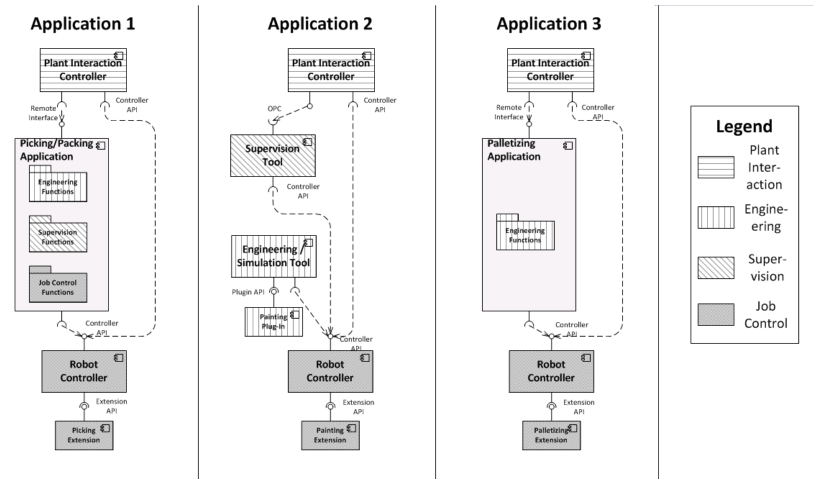


Fig. 4. High-Level Software Architecture for three ABB Robotics PC Applications

a graphical user interface. It communicates with plant interaction controllers via a remote interface and with the robot controllers via the controller API. For this application, the robot controller features a special picking extension. Simulation is not supported for this type of application.

The painting application includes two distinct tools for supervision and engineering/simulation. It does not have additional job control functionality. The engineering and simulation tool has an attached painting plug-in, which tailors the tools for the application domain. The supervision tool communicates with the plant interaction controller via OPC DA (OLE for Process Control Data Access). The supervision tool features a rich and customizable graphical user interface. Additionally, there is a controller extension with special support for painting applications.

The palletizing application mainly provides engineering functionality to set up palletizing jobs. Supervision has to be carried out using the teach pendants or programming the plant interaction controller. There is also no additional job control or simulation functionality. However, there is a robot controller extension specifically for the palletizing domain.

3.3 Extra-functional Requirements

Designing a quality software architecture for robotics PC applications is challenging because of the high extra-functional requirements:

– Availability: Usually, a robot line is part of a larger production process, where a failure of the robots can result in substantial costs. As a particular example, for the picking/packaging application, the job controller functionality must run without interruption. Otherwise, no more targets for the robot controller might be available, which stops the whole production process.

- Scalability: Robot systems are sold to a large variety of customers. Some customers operate small robot cells with single controllers and robots, while other customers run large distributed robot lines with dozens of robots. The architecture must support adapting the application for different installation sizes.
- Maintainability: High maintenance costs should be avoided to keep the applications profitable. Redundant maintenance effort for functionality implemented in multiple applications should be avoided at any costs.
- Time-to-Market: The applications should be adaptable so that new application domains can be supported in a short amount of time. Therefore, reusability of existing assets for new application domains is highly desirable.
- Sustainability: As the robot systems have an expected operation time of more than 10 years, the applications should be ready to cope with technology changes in the future.
- Security: Remotely accessible robot systems need user authentications to avoid being compromised.
- Performance: Once in production, the picking/packing application has to deliver the coordinate batches to the robot controller in time. If the image analysis takes too long, the conveyor tracking mechanism skips item coordinates, which means that items get not processed. Distributing the coordinate batches onto multiple robots and controllers also happens in real time. Static and dynamic load balancing mechanisms must not slow down the robot controllers so that it cannot handle the timing constraints.
- Usability: A common look-and-feel for all ABB Robotics PC applications is desirable so that users can quickly orient themselves when using tools from different application domains.

4 The Solution: Step-wise Evolution

Our project consisted of three phases: reconstructing and documenting the detailed architecture of the picking/placing application (Section 4.1), designing a new remote interface for communication within the architecture (Section 4.2), and finally, designing a new product-line architecture based on identified reusable components from the architecture reconstruction and also including the new remote interface (Section 4.3).

4.1 Architecture Reconstruction and Documentation

As already indicated in Fig. 4, the picking/placing application was perceived as bundling much functionality with limited separation of concerns, which hampered introducing reuse. Therefore, we analysed the architecture of this application in detail in the first phase. Initially, there was no architectural documentation and only limited design documents. First, we reconstructed the architecture, then we documented it, and finally we made suggestions for improvements.

For *architecture reconstruction*, we looked at the application both externally (running different test cases) and internally (analysing the source code). We browsed the code manually to find out how the coarse functionalities listed in Section 3.1 were spread among the code. Furthermore, we used source code analysis tools, such as SourceMonitor [7], Doxygen [8], and SISSy [9] to visualize the static structure and to derive code metrics.

As a result, we recovered a layered architecture as depicted on a high abstraction level in Fig. 5. There are engineering user interfaces and logic as well as runtime user interfaces and logic, the latter including both supervision and and job control functionality. All modules rely on a common object model, which is based on several elementary data types and supports persistency via XML serialization. In total, the application consists of more than 300 KLOC in more than 600 files. Technologically, it is a Win32 application written in C++ with dependencies to Microsoft's Foundation Classes (MFC) and several third-party components.

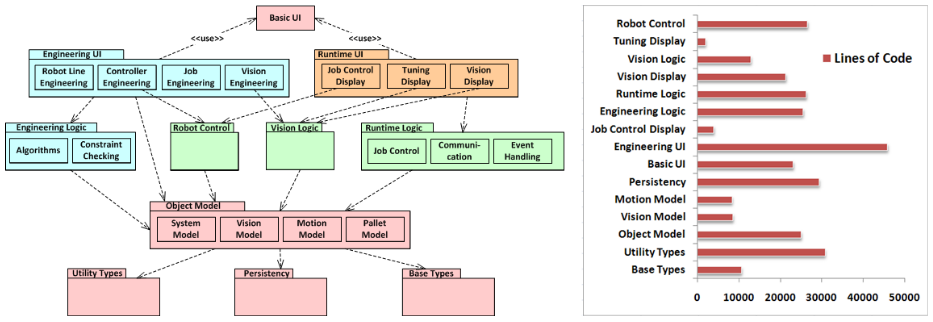


Fig. 5. Picking/Packing Application - Layered Architecture and Code Metrics

While the tools we applied for source code analysis are useful to derive layered structures and bad smells in the code, they provide only limited support for identifying reusable components. Up to now, this is still mainly a manual tasks. Similar tools in this area are Dali, Lattix, or Sotograph. While they can analyse package structures and class dependencies, it is still difficult to locate common functionality spread among multiple packages with these tools. Reverse engineering tools require a more strict software component definition with provided and required interfaces, which are distinct from classes or modules. This way higher level structures could be identified to make components replaceable. A preliminary example for analysing Java code has been presented in [10].

For *architecture documentation*, we used annotated UML diagrams as well as textual descriptions. We used component and package diagrams for a static view on the architecture, as well as sequence diagrams for a dynamic view. Besides the high-level architecture depicted in Fig. 5, we also documented the structure and behaviour on lower levels (e.g., we decomposed included composite components into smaller structures). Our UML models do not directly map to the code structure, but instead visualize higher level structures. In our application domain, UML models are still only used for documentation, not for code generation or analysis of extra-functional properties.

Our suggestions for *architectural improvements* mainly targeted the extra functional requirements modifiability and sustainability. Modifiability requires isolating separated concerns, so that parts of the application become replaceable. It is an important prerequisite for introducing a software product-line approach. Sustainability is especially critical for industrial software application with life-times often longer than typical IT technology life-times.

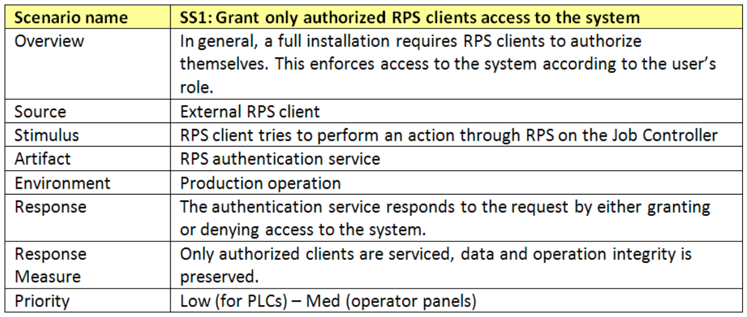
To improve *modifiability* and maintainability, we suggested to enforce the layered structure of the architecture more on the code-level. Modifiability tactics [11], such as localizing modifications by maintaining semantic coherence in the different components and layers were presented. We suggested to factor out more base types from the object model, to restructure the central system package from the source code to adhere to the layered structure, and to isolate UI functionality from the object model, which was not fully decoupled from the higher layers. Additionally, the SISSy tool revealed several problem patterns on the design and implementation level, such as dead imports, cyclic dependencies, god classes, permissive visibility of attributes or methods, or violation of data encapsulation.

To improve *sustainability*, we suggested a step-wise migration of the code-base to the .NET framework. Such a technology change shall result in higher developer productivity due to the higher level APIs of the framework and a modern programming language with C#. The application could reuse .NET platform features, such as the frameworks for user interfaces and persistency. Reliability and security shall be improved via type safe code and a new user authentication mechanism. Besides using newer technologies, this change also prepares the application to incorporate third party components off-the-shelf (COTS), as third party vendors are migrating or have already migrated to the new platform. Therefore, we suggested to replace the number of dependencies to the MFC framework with dependencies to the .NET framework to make the application more portable.

4.2 Extending a Remote Interface

The goal of the second phase of the project was to extend the remote interface of the picking/placing application to allow for more application level services, such as tuning sensor parameters during runtime and remote robot control. The existing remoting interface (called RIS) of the application was mainly used by low-level devices, such as PLCs. The new version of the interface should support higher-level systems such as distributed control systems or customer HMIs. Furthermore, it was required that the interface was compliant to interface standards such as OPC, and regulations by the Organization for Machine Automation and Control (OMAC), which for example requires user authentication.

To formulate the functionality provided by the extended interface, we used UML use cases with textual descriptions. Additionally, we used quality attribute scenarios [11] to specify the extra-functional requirements, such as performance, reliability, and security for the extended interface. They describe the source of a scenario, a stimulus initiating the scenario, the artifact touched by the scenario, environmental conditions, as well as expected responses and response measures. Fig. 6 shows an example for a security scenario of the interface.



Scenario name	SS1: Grant only authorized RPS clients access to the system
Overview	In general, a full installation requires RPS clients to authorize themselves. This enforces access to the system according to the user's role.
Source	External RPS client
Stimulus	RPS client tries to perform an action through RPS on the Job Controller
Artifact	RPS authentication service
Environment	Production operation
Response	The authentication service responds to the request by either granting or denying access to the system.
Response Measure	Only authorized clients are serviced, data and operation integrity is preserved.
Priority	Low (for PLCs) – Med (operator panels)

Fig. 6. Security Scenario for the new Remote Interface

We soon realized that we could not incorporate access by low-level devices and high-level systems into a single interface. Therefore, we subsumed the high-level application services in a new interface called Remote Production Services (RPS), and left the old remote interface intact. The new interface was implemented as a web service based on the Windows Communication Framework (WCF). Additionally, it can be provided as an OPC interface. It allows various functionalities, such as controller management, job management, robot management, user management, logging, and parameter hot tuning. The remoting capabilities of RPS allow external clients to access and control the robot system with an interface at the application level, beyond a generic robot controller interface.

The quality attribute scenarios were helpful in discussions with the stakeholder. We got a better understanding of the application domain because of the quality attribute scenarios. Furthermore, the scenarios helped to define the priorities of the stakeholders, as it was discovered that certain extra-functional properties were only secondary to them. As a result, quality attribute scenarios are currently also used for the specification of a new software architecture.

Later, the new remote interface was an important part of the newly designed product-line architecture.

4.3 Designing a Product-Line Architecture

The goal of the third phase of our project was to design a sustainable software product line architecture based on the applications described in Section 3.2. The design incorporated both the architectural documentation of the picking/placing application from phase 1 and the the newly designed remote interface RPS from phase 2. The main requirements for the product-line were separating different concerns in the applications, increasing reuse, providing a common look-and-feel, and improving maintainability. Additionally the extra-functional requirements stated in Section 3.3 had to be addressed.

Fig. 7 depicts a high-level static view of the new product-line software architecture incorporating all identified core assets. The three existing applications can be deduced as instances from this product-line. New applications for different domains shall be deducible with limited effort. Common functionality has been bundled. For example, there is a single engineering tool with attached application-specific plugins providing a common look-and-feel for engineering. Furthermore, all applications use the same tool for supervision. The extensions for the robot controller have been left untouched. The architecture enables simulation functionality to become available for different domain-specific applications.

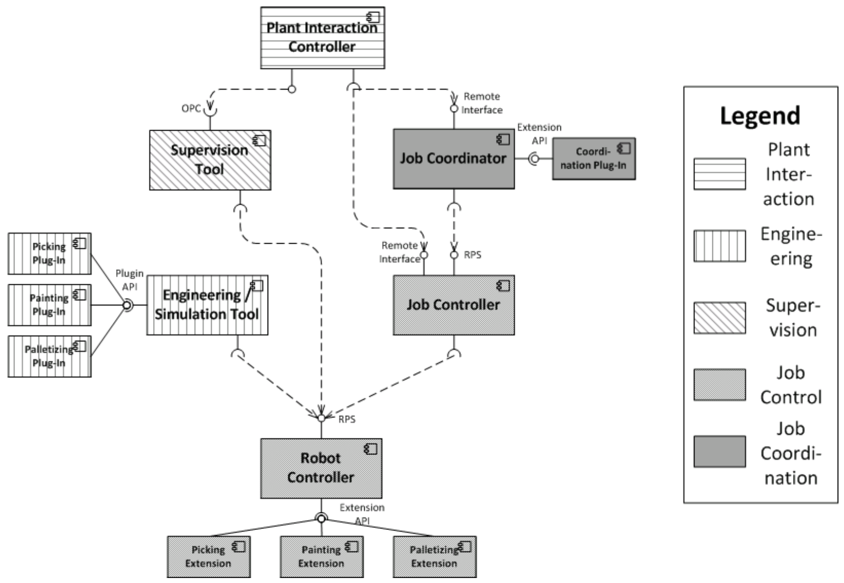


Fig. 7. Product-Line Architecture

For sensor data analysis and item position generation of picking/placing tasks an additional GUI-less job controller component has been extracted from the former picking/placing application. It is an optional component not present in the instances for painting and palletizing applications. We had to align and compare concepts present in the existing applications. For example one application called a configured and running robot application a ”project”, another one called it a ”job” with slightly different properties. Harmonizing the concepts allowed for more reuse and a common look-and-feel of the applications.

The architecture features a new job coordinator component based on .NET technology. It allows coordinating multiple, possibly concurrently running jobs during production. For some custom robot system installations, this functionality had been implemented using PLCs. Opposed to that, the new job coordinator component is based on .NET technology and shall run on Windows nodes. With a special extension API and the .NET framework, it allows users to easily implement customized coordination logic.

Bundling common functionality into reusable components was enabled by the architecture reconstruction and documentation from phase 1. The engineering layer was replaced by the Engineering/Simulation tools based on .NET technology, while the supervision layer was replaced by the Supervision tool. We incorporated the new high-level remote interface RPS from phase 2 into the architecture. It provides a new way of accessing the robot controller with high-level services instead of former low-level commands.

The architecture addresses the extra-functional requirements listed in Section 3.3 as follows:

– Availability: For picking/placing applications, the architecture allows for *multiple* job controllers to analyse sensor data and produce item positions for the robot controller. Formerly, this was a single point-of-failure, as a whole robot line had to stop production once the job controller functionality failed.
– Scalability: The architecture can be flexibly adapted for small and large customers. Small systems might not incorporate the optional job coordinator for synchronizing different jobs. Large systems may additionally attach customer HMIs, which can make use of the RPS interface. They may also use multiple job controllers or even run combined picking/placing and palletizing robot lines.
– Maintainability: As common functionality has been bundled into reusable components, the future maintenance effort for the application should decrease. It is no more necessary to test the same functionality multiple times. Critical functionality, such as the picking/placing job controller has been isolated, so that it can be tested more thoroughly. Using common programming languages and frameworks in the architecture is beneficial to distribute the maintenance tasks to different teams.
– Time-to-market: The product-line architecture features an engineering tool and robot controller, which can be quickly adapted to new application domains via plug-ins. New applications do not have to reprogram basic functionality provided by both platforms (e.g., basic user interfaces and robot control logic).

– Sustainability: The product-line architecture features new components based on the .NET framework, which is expected to ease the impact on technology changes over the course of the robot system life-cycle.
– Security: The RPS interface provides services for user authentication to prevent unwanted remote accesses to a robot system.
– Performance: In case of the picking/placing application, concepts for distributing the sensor data analysis and item position generation to multiple job controller instances have been discussed. This should allow to balance the workload on the available hardware better and enable very large robot lines with vast amounts of sensor data, which were formerly difficult to handle.
– Usability: Through the common engineering tool and the common supervision tool, the look-and-feel of the robotics PC applications for engineers and operators is similar across application domains. With the aligned concepts of the different applications, users can quickly learn the new applications. Furthermore, developers are provided with a common remote interface and several extension APIs, which enable user-customizations with limited development effort.

5 Lessons Learned

While the design of the architecture is specific for the robotics domain and ABB, we have learned some general lessons during the course of the projects. These lessons could stimulate further research from the software architecture community and are thus reported in the following.

On the *technical* side, we learned that in our case no common low-level interfaces both for PLC and PC applications could be provided with reasonable effort. Therefore we split the remote interface for the robot applications to a low-level interface (RIS) to be accessed by controllers and a high-level interface (RPS) with different services on the application level to be accessed by DCSs or ERPs.

Reverse engineering techniques to analyse legacy source code could be improved to better identify common functionality spread within the code of a object-oriented application. While existing source code analysis tools are helpful in capturing the structure of a system, they are limited for identifying reusable functionality to be isolated and bundled into components, if this had not been intended by the architecture beforehand.

Unifying some concepts within the different products (e.g., job and robot line concepts) by introducing small changes gave all stakeholders a better understanding of the different application domains. We found that such a step is an important prerequisite when designing a product-line from legacy applications.

From a *methodological* view point, we found quality attribute scenario and attribute-driven design helpful in determining priorities for different extra-functional properties together with the customers. Both methods helped finding focus when designing the RPS interface and the product-line architecture. Furthermore, we found through a business analysis that the benefits from the architecture investigation in terms of saved future development and maintenance costs largely outweigh its costs.

Some *social* aspects could also be learned from the project. During the design of the product-line architecture, we worked closely with the three development teams of the applications. The existing products and known customer installations had a major impact on our PLA design. The stakeholders of the architecture desired to incorporate all relevant product set-ups into the PLA. A survey of existing products and user customizations was essential to ensure stakeholder support in the PLA.

The development teams were initially hesitant towards the redesign of their applications into a PLA. The emotional bindings towards their established products was an obstacle to get their commitment. We resolved their reluctance by getting the different teams into dialogue and emphasizing their individual benefits from the PLA approach (e.g., more focus on core functionality, less maintenance effort).

Development of a PLA should be aligned with the future plans and development cycles of the individual development teams to ensure their support and make the architecture sustainable. With multiple stakeholders having equal rights to the project, the proposal for the architectural design needed more argumentation and an iterative approach. A champion advocating the benefits of a PLA can speed-up the design process.

6 Related Work

Basic literature on software product-line development has been provided by Clements and Northop [1]. Bass et al. [11] described foundations on software architectures in practise with a special focus on extra-functional requirements. Many industrial case studies as in this paper have been included in the book. Another book on software product-lines has been published by Pohl et al. [12].

In the context of ABB, Mustapic et al. [6] described the software product line architecture of ABB's robotics controller. As described in this paper, it is an open platform, which allows to extend the controller with application specific functionality. The architectural design accounts for fault tolerance and there are methods and tools to assess the real-time properties of different instances of the architecture. Furthermore, Kettu et al. [13] from ABB proposed a holistic approach for architectural analysis of complex industrial software systems, which relies on static and dynamic analysis as well as incorporating documentation, developer interviews, and configuration management data.

In the area of software product line engineering Stoermer et al. [2] presented the MAP approach for mining legacy architecture to derive product lines. O'Brien et al. [3] reports on a case study in this direction using the Dali workbench and reverse engineering techniques to identify components for a product line. Smith et al. [4] describe a method with systematic, architecture-centric means for mining existing components for a product-line.

Numerous software product-lines from the industry have been reported and incorporated into SEI's product-line hall of fame [14]. It includes for example product lines from Bosch for a gasoline system [15] or from Philips for a telecommunication switching system [16]. Hetrick et al. [17] reported on the the incremental return on investment for the transition to a software product-line for Engenio's embedded RAID controller firmware. Deelstra et al. [18] pointed out that deriving individual products from shared software assets in more time-consuming and expensive than expected.

7 Conclusions

Motivated by a survey on ABB robotics software, which found high functional overlap and maintenance costs, we have documented in this paper how we evolved three existing robotics PC applications into a software product line architecture. The PLA addresses various extra-functional properties, such as availability, scalability, performance, and maintainability. The paper has reported several lessons learned from the project, which could stimulate further research.

As next steps, we plan to model the product-line in a formal way and conduct model-driven predictions for extra-functional properties, such as performance, reliability, and maintainability. Creating such models suitable for extrapolating the extra-functional properties to answer sizing and capacity question requires static as well as dynamic analyses techniques. We will assess whether we can predict the impact of system updates or changes based on the models without implementing these changes. These activities shall be conducted in context of the EU FP7 project Q-IMPRESS [19].

Acknowledgements: We thank Peter Weber from ABB Corporate Research for kindly providing us details about the ABB robotics applications survey. Furthermore, we thank all members of the I1 group at ABB Corporate Research Ladenburg for their valuable review comments, which help improving the quality of this paper.

References

1. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley (August 2001)
2. Stoermer, C., O'Brien, L.: MAP - mining architectures for product line evaluations. In: Proceedings of the First Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Amsterdam, Netherlands (August 2001) 35–44
3. O'Brien, L.: Architecture reconstruction to support a product line effort: Case study. Technical Report CMU/SEI-2001-TN-015, Software Engineering Institute (SEI), Carnegie Mellon University (CMU) (July 2001)
4. Smith, D.B., Brien, L.O., Bergey, J.: Using the options analysis for reengineering (oar) method for mining components for a product line. In: SPLC 2: Proceedings

of the Second International Conference on Software Product Lines, London, UK, Springer-Verlag (2002) 316–327
5. Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M., Richter, J.P.: Quasar Enterprise: Anwendungslandschaften service-orientiert gestalten. dpunkt-Verlag (2008)
6. Mustapic, G., Andersson, J., Norstroem, C., Wall, A.: A dependable open platform for industrial robotics - a case study. In: Architecting Dependable Systems II. Volume 3069 of LNCS., Springer (2004) 307–329
7. Campwood Software: SourceMonitor. http://www.campwoodsw.com (Jan 2009)
8. van Heesch, D.: Doxygen: Source code documentation generator tool. http://www.stack.nl/ dimitri/doxygen/
9. Forschungszentrum Informatik (FZI), Karlsruhe: SISSy: Structural Investigation of Software Systems. http://sissy.fzi.de (Jan 2009)
10. Chouambe, L., Klatt, B., Krogmann, K.: Reverse Engineering Software-Models of Component-Based Systems. In Kontogiannis, K., Tjortjis, C., Winter, A., eds.: 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, IEEE Computer Society (April 1–4 2008) 93–102
11. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice (2nd Edition). SEI Series in Software Engineering. Addison-Wesley (2003)
12. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
13. Kettu, T., Kruse, E., Larsson, M., Mustapic, G.: Using architecture analysis to evolve complex industrial systems. In de Lemos, R., Giandomenico, F.D., Gacek, C., Muccini, H., Vieira, M., eds.: Architecting Dependable Systems V: Proceedings of the Workshop on Software Architectures for Dependable Systems (WADS'07). Volume 5135 of LNCS., Springer (2007) 326–341
14. Software Engineering Institute: Product Line Hall of Fame. http://www.sei.cmu.edu/productlines/plp_hof.html (Jan 2009)
15. Steger, M., Tischer, C., Boss, B., Mller, A., Pertler, O., Stolz, W., Ferber, S.: Introducing pla at bosch gasoline systems: Experiences and practices. In: Proc. 3rd Int. Software Product Line Conference (SPLC'04). (2004)
16. Wijnstra, J.G.: Critical factors for a successful platform-based product family approach. In: SPLC 2: Proceedings of the Second International Conference on Software Product Lines, London, UK, Springer-Verlag (2002) 68–89
17. Hetrick, W.A., Krueger, C.W., Moore, J.G.: Incremental return on incremental investment: Engenio's transition to software product line practice. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 798–804
18. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. J. Syst. Softw. **74**(2) (2005) 173–194
19. Q-Impress Consortium: Q-Impress Project Website. http://www.q-impress.eu