Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study

Hongyu Pei Breivold[1], Ivica Crnkovic[2], Rikard Land[2], Magnus Larsson[1]
[1]*ABB Corporate Research, Industrial Software Systems, 721 78 Västerås, Sweden*
{hongyu.pei-breivold, magnus.larsson}@se.abb.com
[2]*Mälardalen University, 721 23 Västerås, Sweden*
{ivica.crnkovic, rikard.land}@mdh.se

Abstract

Evolution of software systems is characterized by inevitable changes of software and increasing software complexity, which in turn may lead to huge maintenance and development costs. For long-lived systems, there is a need to address evolvability (i.e. a system's ability to easily accommodate changes) explicitly in the requirements and early design phases, and maintain it during the entire lifecycle. This paper describes our work in analyzing and improving the evolvability of an industrial automation control system, and presents 1) evolvability subcharacteristics based on the problems in the case and available literature; 2) a structured method for analyzing evolvability at the architectural level - the ARchitecture Evolvability Analysis (AREA) method. This paper includes also the main analysis results and our observations during the evolvability analysis process in the case study. The evolvability subcharacteristics and the method should be generally applicable, and they are being validated within another domain at the time of writing.

1. Introduction

Studies indicate that more than 50% of the total life cycle cost is spent after the initial development [18]. Therefore, it becomes essential to cost-effectively carry out software evolution. In order to prolong the productive life of a software system, the need to explicitly address software evolvability is becoming recognized [6]. There are examples of industrial systems with a lifetime of 20-30 years. These systems are subject to and may undergo a substantial amount of evolutionary changes, e.g. shifting business and organizational goals, software technology changes, software systems merge due to organizational changes [16], demands for distributed development, system migration to product line architecture, etc. The evolution problems we have observed came from various cases in industrial context, where evolvability was identified as a very important quality attribute that must be maintained. In order to preserve and improve evolvability, we need to (i) analyze the system with respect to evolvability; and (ii) perform architectural transformation. It is generally acknowledged that the software's architecture holds a key to the possibility to implement changes in an efficient manner [1]. Therefore, in this paper, we analyze evolvability at the architecture level and identify the evolvability subcharacteristics of interest in an industrial case study, where a large automation control system at ABB was evolved from a monolithic architecture towards a product line. We present our experiences of the development of the product line architecture in the form of a general method, which we have constructed from data in the manner of *grounded theory research* [25]. In addition, the risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences.

The remainder of this paper is structured as follows. Section 2 describes the context of the case study. Section 3 presents our architecture evolvability analysis method - AREA. Section 4 presents the case study, in which the method was applied to analyze, evaluate and improve the software architecture of the automation controller software system. Section 5 discusses the experiences we gained through the case study. Section 6 reviews related work. Section 7 concludes the paper.

2. Context of the Case

This section presents the case to motivate evolvability analysis and describe seven evolvability subcharacteristics from the case perspective.

2.1 Motivating Evolvability Analysis

The case study was based on a large automation control system at ABB and focused on the latest generation of the controller. The controller software consists of more than three million lines of code written in C/C++ and uses a complex threading model, with support for a variety of different applications and devices. It has grown in size and complexity, as new features and solutions have been added to enhance functionality and to support new hardware, such as devices, I/O boards and production equipment. Such a complex system is difficult to maintain. It is also

important and considerably more difficult to evolve. Due to different measures such as organizational and lifecycle process improvements, the system keeps the maintainability, but the evolvability becomes more difficult since the increased complexity in turn leads to decreased flexibility, resulting in problems to add new features. Consequently, it becomes costly to adapt to new market demands and penetrate new markets.

Our particular system is delivered as a single monolithic software package, which consists of various software applications developed by distributed development teams. These applications aim for specific tasks in painting, welding, gluing, machine tending and palletizing, etc. To keep the integration and delivery process efficient, the initial architectural decision was to keep the deployment artifact monolithic. The complete set of functionality and services is present in every product even though not everything is required in the specific product. As the system grew, it became more difficult to ensure that the modifications of specific application software do not affect the quality of other parts of the software system. As a result, it became difficult and time-consuming to modify software artifacts, integrate and test products. To continue exploiting the substantial software investment made and to continuously improve the system for longer productive lifetime, it has become essential to explicitly address evolvability, since software evolvability is a fundamental element for increasing strategic and economic value of the software [28]. The inability to effectively and reliably evolve software systems means loss of business opportunities [2].

2.2 Evolvability Subcharacteristics from Case Perspective

In our previous work [21], we have identified subcharacteristics that are of primary importance for an evolvable software system. Definitions and detailed explanations of evolvability subcharacteristics are provided in [21]. The derivation of evolvability subcharacteristics are based on survey and analysis of literatures (see related work section), problems we have observed and experiences from several earlier case studies. We do not exclude the possibilities that other domains or cases might have slightly extended set of subcharacteristics. Each subcharacteristic is explained below in conjunction with the case.

Analyzability The release frequency of the controller software is twice a year, with around 40 various new major requirements that need to be implemented in each release. These requirements have impact on different attributes of the system, and the possible impact must be analyzed before the implementation of the requirements. This requires that the software system must have the capability to be analyzed and explored in terms of the impact to the software by introducing a change.

Architectural Integrity A strategy for communicating architectural decisions that we found out from various case studies was to appoint members of the core architecture team as technical leaders in the development projects. However, this strategy although helpful to certain extent, did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions, resulting in unconscious violation of architectural conformance. This may lead to evolvability degradation in the long run. Therefore, it is important to record rationale for each design decision, strategy and architectural solution.

Changeability Due to the monolithic characteristic of the controller software, modifications in certain parts of the software package lead to some ripple effects, and requires recompiling, reintegrating and retesting of the whole system. This results in inflexibility of patching and customers have to wait for a new release even in case of corrective maintenance and configuration changes. Therefore, it is strongly required that the software system must have the ease and capability to be changed without negative implications or with controlled implications to the other parts of the software system.

Portability The current controller software supports VxWorks and Microsoft Windows NT. There is a need of openness for choosing among different operating system (OS) vendors, e.g. Linux and Windows CE, and possibly new OS in the future.

Extensibility The current controller software supports around 20 different applications that are developed by several distributed development centers around the world. To adapt to the increased customer focus on specific applications and to enable establishment of new market segments, the controller, like any other software systems, must constantly raise the service level through supporting more functionality and providing more features [4], while keeping some important extra-functional properties, such as performance, or reliability.

Testability The controller software exposed huge number of public interfaces which resulted in tremendous time merely on interface tests. One task was therefore to reduce the public interfaces to around 10% of the original public interfaces. Besides, due to the monolithic characteristic, error corrections in one part of the software requires retesting of the whole system. One issue was therefore to investigate the feasibility of testing only modified parts.

Domain- specific attributes The controller software has critical real-time calculation demands. It is also expected to reduce the base software code size and runtime footprint.

3. Overview of the ARchitecture Evolvability Analysis (AREA) Method

The steps that we performed in the case are divided into three main phases as shown in Figure 1.

Phase 1*: Analyze the implications of change stimuli on software architecture.*

This phase analyzes the architecture for evolution and understands the impact of change stimuli on the current architecture. Software evolvability concerns both business and technical issues [29], since the stimuli of changes come from both perspectives, e.g. environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software structures and/or functionality.

Step 1.1: *Identify potential requirements in the software architecture.*

Any change stimulus results in a collection of potential requirements that the software architecture needs to adapt to. The aim of this step is to extract these requirements that are essential for software architecture enhancement so as to cost-effectively accommodate to change stimuli. Architecture workshops can be conducted, where the stakeholders discuss and identify the potential architecture requirements. Each requirement is concretized with a collection of identified refined activities. Afterwards, each identified requirement must be checked against the evolvability subcharacteristics so as to ensure the consistency and completeness.

Step 1.2: *Prioritize potential requirements in the software architecture.*

In order to establish a basis for common understanding of the architecture requirements among stakeholders within the organization, all the potential requirements identified from the first step need to be prioritized. We do not propose any general criteria for requirement prioritization that apply to all the software systems evolution, since the criteria might be different from case to case depending on factors such as development and organizational constraints, the probability of potential requirements becoming mandatory requirements that the architecture must adapt to, etc.

Phase 2*: Analyze and prepare the software architecture to accommodate change stimuli and potential future changes.*

This phase focuses on the identification and improvement of the components that need to be refactored.

Step 2.1: *Extract architectural constructs related to the respective identified requirement.*

We mainly focus on architectural constructs that are related to each identified requirement. In order for the architecture to allow changes in the software without compromising software integrity and to evolve in a controlled way, documentation of architectural decisions and their rationale play a key role.

Step 2.2: *Identify refactoring components for each identified requirement.*

In this step, we identify the components that need refactoring in order to fulfill the prioritized requirements.

Step 2.3: *Identify and assess potential refactoring solutions from technical and business perspectives.*

Refactoring solutions are identified and design decisions are taken in order to fulfill the requirements derived from the first phase. The change propagation of the effect of refactoring need to be considered and provided as an input to the business assessment, estimating the cost and effort on applying refactorings. In some cases, the refactoring of a certain component is straightforward if we know how to refactor with only local impact. When the implementation is uncertain and might affect several subsystems or modules, prototypes need to be made to investigate the feasibility of potential solutions as well as the estimation of implementation workload. As part of this step, an assessment regarding the compatibility of the refactoring solutions and rationale with earlier made design decisions is made to ensure architectural integrity.

Step 2.4: *Define test cases.*

New test cases that cover the affected component, modules or subsystems need to be identified.

Phase 3*: Finalize the evaluation.*

In this phase, the previous results are incorporated, analyzed and structured into a collection of documents.

Step 3.1*: Analyze and present evaluation results.*

The evaluation results include (i) the identified and prioritized requirements on the software architecture; (ii) the identified components/modules that need to be refactored for enhancement or adaptation; (iii) refactoring investigation documentation which describes the current situation, rationale and solutions to each identified candidate that need to be refactored, including estimated workload; (iv) test scenarios; and (v) impact analysis on evolvability.

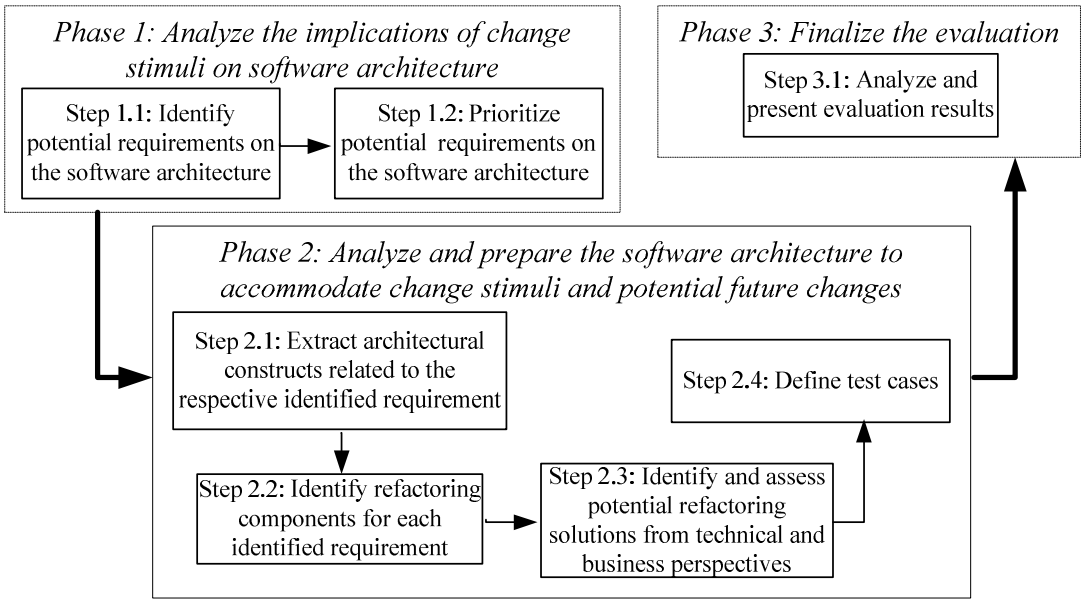

Figure 1. The steps of the ARchitecture Evolvability Analysis (AREA) method

4. Applying the AREA Method

The main focus of the analysis in our case was to assess how well the architecture would support potential forthcoming requirements and understand their impact. Through the analysis process, we identified potential flaws and defined an evolution path of the software system. The identification and analysis of the architectural requirements was performed by the architecture core team which consists of 6-7 persons. It was a continuous maturation process from the first vision to concrete activities that took approximately one calendar year including analysis, identification of architecture evolution path and partial refactoring. 2-3 persons from the architecture core team identified the refactoring solution proposals for the components in the *Basic Services* subsystem. These proposals were discussed with the main technical responsible persons and architects, documented as evolution path for the architecture and transferred further to the implementation teams.

4.1 Phase 1 - Step 1.1: Identify potential requirements on the software architecture

The change stimuli to the controller software came from the following emerging critical issues related to software evolution: (a) time-to-market requirements, such as building new products for dedicated market within short time; (b) improvement of software system evolvability; and (c) increased ease and flexibility of distributed development of products in combination with the diversity of application variants. We list below the main potential architecture requirements that were identified from the change stimuli. The refined activities for each requirement are presented as well.

R1. Improved modularization of architecture.

a) Enable the separation of layers within the controller software: (i) a kernel which comprises of components that must be included by all application variants; (ii) common extensions which are available to and can be selected by all application variants; and (iii) application extensions which are only available to specific application variants.
b) Investigate dependencies between the existing extensions.

R2. Reduced architecture complexity.

a) Define interfaces and reduce public interface calls.
b) Add support for task isolation and task management.

R3. Enable distributed development of extensions with minimum dependency.

a) Build the application-specific extensions on top of the base software (kernel and common extensions) without the need of modification to the internal base source code.
b) Package the base software into SDK (Software Development Kit), which provides necessary interfaces, tools and documentation to support distributed application development and separate release cycles of the SDK and application-specific extensions.

R4. Improved portability.

a) Investigate portability across target operating system platforms and across hardware platforms.

R5. Impact on product development process.

a) Investigate the implications of software restructuring on product integration and testing.

R6. Minimized software code size and runtime footprint.

a) Investigate enabling mechanisms, e.g. properly partitioning functionality.

The above architecture requirements should be checked against the evolvability subcharacteristics to justify whether the realization of each requirement would lead to an improvement of the subcharacteristics (or possibly a decrease, which would then require a tradeoff decision), as summarized in Table 1. Besides, the choice of component refactoring and implementation solution proposals for fulfilling each requirement might cause tradeoffs against some other subcharacteristics, as detailed in section 4.7.

Table 1. Mapping between evolvability subcharacteristics and architecture requirements

Subcharacteristics	Requirements
Analyzability	R1. Improved modularization of architecture. R2. Reduced architecture complexity.
Architectural Integrity	not related to any particular architectural requirement, but rather to whether the architectural choices and rationale for handling these requirements are documented
Changeability	R1. Improved modularization of architecture. R2. Reduced architecture complexity.
Extensibility	R3. Enable distributed development of extensions with minimum dependency.
Portability	R4. Improved portability.
Testability	R5. Impact on product development process.
Domain-specific attributes	R6. Minimized software code size and runtime footprint.

4.2 Phase 1 - Step 1.2: Prioritize potential requirements on the software architecture

Due to the monolithic characteristics of the architecture, the individual products are burdened with functionalities and components that are not necessary for the specific individual products. Accordingly, the main idea was to apply the product line approach, transform the existing system into reusable components that can form the core of the product-line infrastructure, and separate application-specific extensions from the base software. With the consideration of not disrupting the ongoing development projects, the criteria for requirement prioritization were: (i) enable building of existing types of extensions after refactoring and architecture restructuring; (ii) enable new extensions and simplify interfaces that are difficult to understand and may have negative effects on implementing new extensions. Based on these criteria, R1, R2 and R3 were prioritized potential architectural requirements.

4.3 Phase 2 - Step 2.1: Extract architectural constructs related to the respective identified requirement

Over years of development, a lot of functionality has been added to the system to support new requirements. It becomes easy to unconsciously violate the original good design decisions. To prevent this, it is important to extract design decisions and rationale through documentation of architectural constructs. In this way, potential architectural flaws can be discovered. For instance, in the case study, some implementation violations were discovered, such as improper use of conditional compilation in case of environment changes, direct access to OS native APIs, etc. Additional efforts have been put to provide training, guidelines/rules and code examples for software developers in writing code and using tactics that enable the achievement of a certain quality characteristic. We exemplify with R3 and extract architectural constructs in form of the original coarse-grained architecture as depicted in Figure 2.

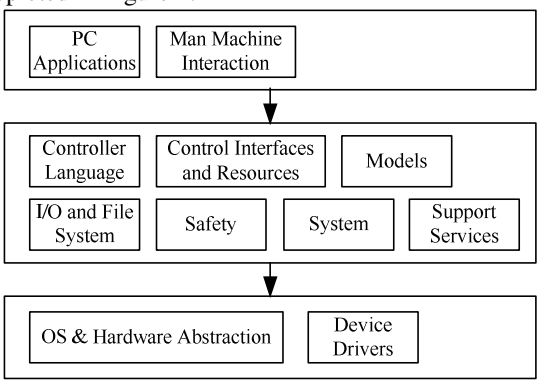

Figure 2. A conceptual view of the original software architecture

The lower layer provides an interface to the upper layer and allows the source code of the upper layer to be used on different hardware platforms and operating systems. The main problem with this software architecture was the existence of tight coupling among some components that reside in different layers. This led to additional work required at a lower level to modify some existing functionality and add support for new functionality in various applications. For instance, the system is required to perform certain tasks during start-up and shutdown in the controller. Some routines for handling such tasks had to be hard-coded, i.e. the application developers had to edit in the source code of e.g. *Support Services* subsystem in the lower layer, which is developed by another group of developers. Accordingly, source code updates had to be done not only on the application level, but through several layers, several subsystems and components.

Recompilation of the whole code base was required. This required that application developers need to have a thorough knowledge of the complete source code. It also constituted a bottleneck in the effort to enable distributed application development.

4.4 Phase 2 - Step 2.2: Identify refactoring components for each identified requirement

To cope with the architectural problems identified in the previous step, the strategy of separate concerns need to be applied to isolate the effect of changes to parts of the system [11], i.e. separate the global functions from the hardware, and separate application-specific functions from generic and basic functions as illustrated in Figure 3.

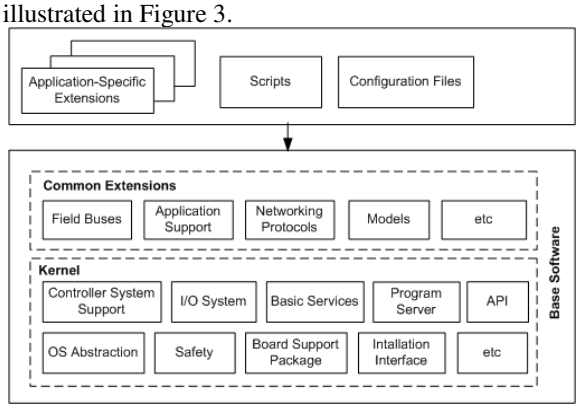

Figure 3. A revised conceptual view of the software architecture

Accordingly, some components need to be adapted and reorganized to enable the architecture restructuring, e.g. some components within the low-level *Basic Services* subsystem for resource allocations, including semaphore ID management component, memory allocation management component to separate functionality from resource management and to achieve the build- and development-independency between the kernel and extensions.

4.5 Phase 2 - Step 2.3: Identify and assess potential refactoring solutions from technical and business perspectives

Due to space limitations and company confidentiality, we exemplify with one component example (inter-process communication component) that needed to be refactored to represent and illustrate for the many various discussions and solutions that occurred during the analysis. We discuss in terms of the following views: (i) problem description: the problem and disadvantages of the original design of the component; (ii) requirements: the new requirements that the component needs to fulfill; (iii) improvement solution: the architectural solution to design problems; (iv) rationale and architectural consequences: the rationale of the solution proposal and architectural implications of the deployment of the component on quality attributes; and (v) estimated workload: the estimated workload for implementation and verification.

4.5.1 Inter-Process Communication. This component belongs to *Basic Services* subsystem and it includes mechanisms that allow communication between processes, such as remote procedure calls, message passing and shared data.

Problem Description. All the slot names and slot IDs that are used by the kernel and extensions are defined in a C header file in the system. The developers have to edit this file to register their slot name and slot ID, and recompile. Afterwards, both the slot name and slot ID have to be specified in the startup command file for thread creation. There is no dynamic allocation of connection slot.

Requirements. The refactoring of this component is related to R3. It should be possible to define and use IPC slots in common extensions and application extensions without the need to edit the source code of the base software and recompile. The mechanism for using IPC from extensions must be available also in the kernel, to facilitate move of components from kernel to extensions in the future.

Improvement Solution. The slot ID for extension clients should not be booked in the header file. Extensions should not hook a static slot ID in the startup command file. The command attribute dynamic slot ID should be used instead. The IPC connection for extension clients will be established dynamically through the `ipc_connect` function as shown in Figure 4. It will return a connection slot ID when no predefined slot ID is given. An internal error will be logged at startup if a duplicate slot name is used.

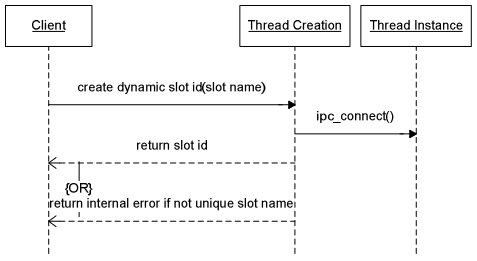

Figure 4. The inter-communication component after refactoring

Rationale and Architectural Consequences. The revised IPC component provides efficient resource booking for inter-process communication and enables encapsulation of IPC facilities. Accordingly, distributed development of extensions utilizing IPC

functionality is facilitated. The use of dynamic inter-process communication connections addressed resource limitations for IPC connection. In this way, limited IPC resources are used only when the processes are communicating. However, the use of IPC mechanisms requires resources, which are limited on a real-time operating system. Therefore, the overheads due to resource description processing may be the offset against efficiency [22], since the overall real-time performance may be degraded if the cost of creating and destroying IPC connections is too high.

Estimated Workload. It was estimated around 2 man weeks which includes the IPC component refactoring and moving IPC client from kernel to extension.

4.6 Phase 2 - Step 2.4: Define test cases

The corresponding test cases were derived based on the selected improvement solution proposal to each component that needed refactoring. For instance, the architectural test cases for the IPC component are given by the ThreadCreation class creating dynamic slot ID, as shown in Figure 5.

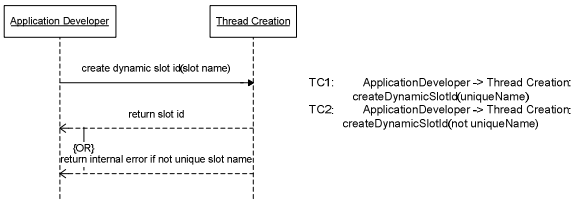

Figure 5. Test cases for IPC management component

4.7 Phase 3 - Step 3.1: Present evaluation results

In this step, the implications of the potential improvement strategies and evolution path of the software architecture are analyzed with respect to the evolvability subcharacteristics as illustrated in Table 2.

Table 2. Impacts of the IPC component on evolvability subcharacteristics (+ positive impact, - negative impact)

	Consequences of changing IPC component
Analyzability	– due to less possibility of static analysis since definitions are defined dynamically
Architectural Integrity	+ due to documentation of specific requirements, architectural solutions and consequences
Changeability	+ due to the dynamism which makes it easier to introduce and deploy new slots
Portability	+ due to improved abstraction of Application Programming Interfaces (APIs) for IPC
Extensibility	+ due to encapsulation of IPC facilities and dynamic deployment
Testability	No impact
Domain-specific attributes	+ resource limitation issue is handled through dynamic IPC connection – due to introduced dynamism, the system performance could be slightly reduced

5. Reflections

This section summarizes our observations and experiences of applying AREA.

5.1 Experiences

By applying AREA method, we have improved the capability in being able to on forehand understand and analyze systematically the impact of a change stimulus. This, in turn, helps us to prolong the evolution stage [2]. Besides, we list below two observations that concern visible improvements in the organization. They were perceived and informally reported by the stakeholders themselves.

Documentation of architecture is improved, including the architecture's evolution path. Architecture transformation and suggestions for refactoring solutions were part of the analysis process. This was performed by the architecture core team. As a result of the analysis and refactoring activities, the documentation of design and implementation solution proposals has been improved. The final refactoring analysis investigation report was distributed for inspection and was approved after a few iterations. This document served as an input and blueprint to the implementation teams. In this way, the architecture core team and implementation teams shared the same view on the evolution path of the software architecture.

High-level business goals lead to architectural requirements. In the case study, the potential requirements on the architecture were derived from the high-level business goals through the first phase, where the potential requirements on the architecture were identified based on the change stimuli. Such derivation provides an understanding on how the intended software system and its evolving artifacts reflect and contribute to the strategic goals. Together with the documentation of architecture evolution path, it would enrich architectural models and facilitate the traceability of software architecture evolution back to the various business constraints and assumptions [15].

5.2 Suggestions

Due to continuously changing requirements and evolutions of new technologies, the software architecture needs to be evolvable to cost-effectively accommodate changes. Thus, we suggest routine evolvability analysis that should be applied as an integral part during the whole software lifecycle.

Another remark is that the process of making the impact analysis of component refactoring in terms of estimated workload was not an easy task. One principle that was applied during the component refactoring process was to preserve the external behavior of the system despite the number of changes to the code. This

required a comprehensive understanding of the dependencies among different components within different subsystems. Good tool support that assists in impact analysis of ripple effects would be helpful.

6. Related Works

To evaluate evolvability, Ramil and Lehman proposed metrics based on implementation change logs [23] and computation of metrics using the number of modules in a software system [17]. Another set of metrics is based on software life span and software size [27]. In [26], a framework of process-oriented metrics for software evolvability was proposed to intuitively develop evolvability metrics and to trace the metrics back to the evolvability requirements based on the NFR framework [5]. However, they do not explicitly address the evolvability analysis at architectural level. The best known quality models e.g. McCall [20], Boehm [3], FURPS [10], ISO 9126 [12] and Dromey [9], do not explicitly address evolvability. An approach was described in [19] to measure software architecture's quality characteristics through identified key use cases, based on the customization of the ISO 9126 standard. An ontological basis which allows for the formal definition of a system and its change at the architectural level is presented in [24].

Kolb et al. [14] presented a case study in refactoring an existing software component for reuse in a product line using the PuLSE approach. Experiences of using various assessment techniques for software architecture evaluation were presented in [8], where scenario-based assessment, software performance assessment and experience-based assessment were addressed. The scenario-based methods such as ATAM [7] would require quite a number of evolvability scenarios (to address and cover each of the seven subcharacteristics); a more important limitation is that while scenarios are concrete anticipated events in the system life-time, evolvability might concern high-level business requirements at an abstract level which calls for some more general type of analysis to identify implications on software architecture and corresponding evolution path.

7. Concluding Remarks

In this paper, we described an analysis of a complex industrial control system, driven by the need to improve its evolvability. A set of evolvability subcharacteristics were described from the case perspective: analyzability, architectural integrity, changeability, portability, extensibility, testability and domain-specific attributes. In addition, an architectural evolvability analysis method (designated as AREA method) was applied to the complex industrial system. The method made the architecture requirements, corresponding design decisions, rationale and architecture evolution path more explicit, better founded and documented, and the resulting documentation of refactoring improvement proposals was widely accepted by the involved stakeholders. The analysis results served as an input and blueprint to the implementation teams. We want to point out that the commitment from the organization to perform such a total restructuring of a large system signifies the importance of software evolvability.

The AREA method is presently being applied in another case within ABB, through which we plan to further refine and validate the method. Another aspect that we are considering is to apply the method to address evolvability explicitly in the early design phase of a new development effort, since software architecture that is capable of accommodating change must be specifically designed for change [13].

References

[1] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison- Wesley. (2003)

[2] Bennett, K., Rajlich, V.: Software Maintenance and Evolution: a Roadmap. The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press. (2000)

[3] Boehm, B.W. et al. : Characteristics of Software Quality. Amsterdam. North-Holland. (1978)

[4] Bosch, J.: Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach. Addison-Wesley. (2000)

[5] Chung, L. et al.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers. (2000)

[6] Ciraci, S., Broek, P.: Evolvability as a Quality Attribute of Software Architectures. ERCIM Workshop on Software Evolution. (2006)

[7] Clements, P., Kazman, R. and Klein, M. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley. (2002)

[8] Del Rosso, C.: Continuous Evolution Through Software Architecture Evaluation: a Case Study. Journal of Software Maintenance and Evolution: Research and Practice. (2006)

[9] Dromey, G.: Cornering the Chimera. IEEE Software (January): 33-43. (1996)

[10] Grady, R., Caswell, D.: Software Metrics: Establishing a Company-Wide Program. Englewood Cliffs, NJ, PrenticeHall. (1987)

[11] Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley. (2000)

[12] ISO/IEC 9126-1. International Standard. Software Engineering: Product Quality, Part 1: Quality Model. (2001)

[13] Isaac, D., McConaughy, G.: The Role of Architecture and Evolutionary Development in Accommodating Change. Proc. NCOSE'94. (1994)

[14] Kolb, R., Muthig, D., Patzke, T., Yamauchi, K.: Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study. Journal of Software Maintenance and Evolution: Research and Practice. (2006)

[15] Lago, P., van Vliet, H.: Explicit Assumptions Enrich Architectural Models. ICSE. (2005)
[16] Land, R., Crnkovic, I.: Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection. Journal of Information and Software Technology, vol 49, nr 5, p419-444, Elsevier, September. (2006)
[17] Lehman, M.M, Ramil, J.F. et al.: Metrics and Laws of Software Evolution: The Nineties View. IEEE Computer Press, pp 20-32. (1997)
[18] Lientz, B., Swanson, E.: Software Maintenance Management. Addison-Wesley. Reading. MA. (1980)
[19] Losavio, F. et al.: ISO Quality Standards for Measuring Architectures. The Journal of Systems and Software. (2004)
[20] McCall, J.A., Richards, P.K., Walters, G.F.: Factors in Software Quality. National Technical Information Service. (1977)
[21] Pei Breivold, H., Crnkovic, I., Eriksson, P. J.: Analyzing Software Evolvability. Accepted at 32nd COMPSAC. (2008)
[22] Quecke, G., Ziegler, W.: Mesch - an approach to resource management in a distributed environment. In Proc. of the First IEEE/ACM International Workshop on Grid Computing. Springer-Verlag, pp. 47–54. (2000)
[23] Ramil, J.F., Lehman, M.M.: Metrics of Software Evolution as Effort Predictors - A Case Study. Proc. ICSM. (2000)
[24] Rowe, D., Leaney, J.: Evaluating Evolvability of Computer Based Systems Architectures – an Ontological Approach. Proc. of International Conference and Workshop on Engineering of Computer-Based Systems. (1997)
[25] Strauss, A. and Corbin, J. M.: Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory (2nd edition), ISBN 0803959400, Sage Publications, 1998.
[26] Subramanian, N., Chung, L.: Process-Oriented Metrics for Software Architecture Evolvability. Proc. IWPSE. (2002)
[27] Tamai, T., Torimitsu, Y.: Software Lifetime and its Evolution Process over Generations. Proc. ICSM. (1992)
[28] Weiderman, N.H. et al.: Approaches to Legacy Systems Evolution. Technical Report CMU/SEI-97-TR-014. (1997)
[29] Yang, H., Ward, M.: Successful Evolution of Software Systems. Artech House Publishers. London. (2003)