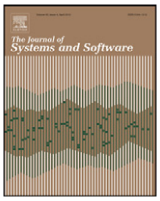
The Journal of Systems and Software 85 (2012) 2574–2592

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



Software architecture evolution through evolvability analysis

Hongyu Pei Breivold[a,*], Ivica Crnkovic[b], Magnus Larsson[a]

[a] ABB Corporate Research, Industrial Software Systems, 721 78 Västerås, Sweden
[b] Mälardalen University, 721 23 Västerås, Sweden

ARTICLE INFO

Article history:
Received 11 November 2011
Received in revised form 29 May 2012
Accepted 29 May 2012
Available online 28 June 2012

Keywords:
Software architecture evolution
Software evolvability

ABSTRACT

Software evolvability is a multifaceted quality attribute that describes a software system's ability to easily accommodate future changes. It is a fundamental characteristic for the efficient implementation of strategic decisions, and the increasing economic value of software. For long life systems, there is a need to address evolvability explicitly during the entire software lifecycle in order to prolong the productive lifetime of software systems. However, designing and evolving software architectures are the challenging task. To improve the ability to understand and systematically analyze the evolution of software system architectures, in this paper, we describe software architecture evolution characterization, and propose an architecture evolvability analysis process that provides replicable techniques for performing activities to aim at understanding and supporting software architecture evolution. The activities are embedded in: (i) the application of a software evolvability model; (ii) a structured qualitative method for analyzing evolvability at the architectural level; and (iii) a quantitative evolvability analysis method with explicit and quantitative treatment of stakeholders' evolvability concerns and the impact of potential architectural solutions on evolvability. The qualitative and quantitative assessments manifested in the evolvability analysis process have been applied in two large-scale industrial software systems at ABB and Ericsson, with experiences and reflections described.

1. Introduction

Change is an essential factor in software development, as software systems must respond to evolving requirements, platforms and other environmental pressures (Godfrey and German, 2008). It has long been recognized that, for long life industrial software, the greatest part of lifecycle costs is invested in the evolution of software to meet changing requirements (Bennett, 1996). To keep up with new business opportunities, the need to change software on a constant basis with major enhancements within a short timescale puts critical demands on the software system's capability of rapid modification and enhancement. Lehman et al. (2000) describe two perspectives on software evolution: "*what* and *why*" versus "*how*". The "*what* and *why*" perspective studies the nature of the software evolution phenomenon and investigates its driving factors and impacts. The "*how*" perspective studies the pragmatic aspects, i.e., the technology, methods and tools that provide the means to control software evolution. In this research, we focus on the "*how*" perspective of software evolution.

The term evolution reflects "*a process of progressive change in the attributes of the evolving entity or that of one or more of its constituent elements*" (Madhavji et al., 2006). Specifically, software evolution relates to how software systems change over time (Yu et al., 2008). One of the principle challenges in software evolution is therefore the ability to evolve software over time to meet the changing requirements of its stakeholders (Nehaniv and Wernick, 2007), and to achieve cost-effective evolution. In this context, software evolvability has emerged as an attribute that "*bears on the ability of a system to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity*" (Rowe et al., 1994).

The ever-changing world makes evolvability a strong quality requirement for the majority of software systems (Borne et al., 1999; Rowe and Leaney, 1997). The inability to effectively and reliably evolve software systems means the loss of business opportunities (Bennett and Rajlich, 2000). Based on our experiences and observations from various cases in industrial contexts (Breivold et al., 2008b; Del Rosso and Maccari, 2007; Land and Crnkovic, 2007), we have noticed that industry starts to have serious considerations with respect to evolvability beyond maintainability. From these studies, we also witness examples of different industrial systems that have a lifespan of 10–30 years and are continuously changing. These systems are subject to and may undergo

* Corresponding author. Tel.: +46 21 323243; fax: +46 21 323212.
E-mail addresses: hongyu.pei-breivold@se.abb.com (H.P. Breivold), ivica.crnkovic@mdh.se (I. Crnkovic), magnus.larsson@se.abb.com (M. Larsson).

a substantial amount of evolutionary changes, e.g., software technology changes, system migration to product line architecture, ever-changing managerial issues such as demands for distributed development, and ever-changing business decisions driven by market situations. Software systems must often reflect these changes to adequately fulfill their roles and remain relevant to stakeholders (Mens et al., 2010a). Evolvability was therefore identified in these cases as a very important quality attribute that must be continuously maintained during their lifecycle. In this paper, we distinguish evolvability from maintainability, because they both exhibit their own specific focus, e.g., type of change. Evolvability is mostly concerned with coarse-grained, long-term, higher-level, radical functional or structural enhancements or adaptations, whereas maintainability is mostly concerned with fine-grained, short-term, localized changes (Cai and Huynh, 2007; Weiderman et al., 1997). As software evolvability is a fundamental element for the efficient implementation of strategic decisions, and the increasing economic value of software (Cai and Huynh, 2007; Weiderman et al., 1997), for such long life systems, there is a need to address evolvability explicitly during the entire lifecycle and thus prolong the productive lifespan of software systems. Surprisingly there are rather few publications that explicitly address evolvability characterization and evolvability assessment (Breivold et al., 2011). A systematic evolvability assessment requires answers to the following questions: (a) what characterizes evolvability of a software system? and (b) how to assess evolvability of a software system in a systematic way? In similar approaches, models and methods (see Section 7) there is a lack of explicit addressing of evolvability.

Our research focuses on the analysis of software evolution at an architectural level for two reasons. Firstly, the foundation for any software system is its architecture, which allows or precludes most of the quality attributes of the system (Clements et al., 2002) and provides the basis for software evolution analysis (Mens et al., 2010b). Secondly, the architecture of a software system not only describes its high level structure and behavior, but also includes principles and decisions that determine the system's development and its evolution (Bengtsson et al., 2004). In this sense, software architecture exposes the dimensions along which a system is expected to evolve (Garlan, 2000) and provides the basis for software evolution (Medvidovic et al., 1998). We also recognize the tight relationship between architecture, organization, business or development processes as indicated in Larsson et al. (2007), that a change along one of these dimensions will require a review of the others in the light of the proposed change. However, because the relationship and impact between these different dimensions during software evolution is a research topic by itself, in this research, we study the evolution of software architecture, and investigate ways to support this evolution.

The main objective of our research is to improve the ability to understand and systematically analyze the evolution of software architectures. Specifically, we state the following research questions:

(1) What software characteristics are necessary to constitute an evolvable software system?
(2) How does one assess the evolvability of a software system in a systematic manner when evolving the system architecture?

In this paper, we describe and make contributions to the following aspects:

(1) Propose a software evolvability model and identify characteristics that are necessary for the evolvability of a software system.
(2) Define the evolvability assessment process, and propose qualitative and quantitative software evolvability assessment methods.
(3) Demonstrate the application of the evolvability model and analysis methods through two case studies of the architecture evolution of large-scale software systems in different industrial settings.

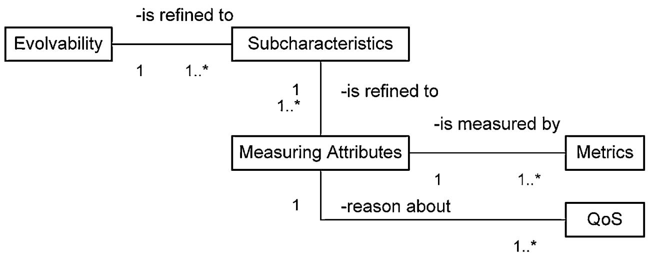


Fig. 1. Software evolvability model.

The remainder of the article is structured as follows. Section 2 describes our software evolvability model which is the basis for the proposed evolvability analysis. Section 3 presents the general software evolvability analysis process along with detailed descriptions of the qualitative and quantitative architecture evolvability analysis methods. Section 4 presents an industrial case study in which the qualitative method was applied to analyze and improve the software architecture of a complex industrial automation control system at ABB. Section 5 presents an industrial case study in which the quantitative method was applied to analyze the potential evolution path of the software architecture of a mobile network node system at Ericsson. In both cases, the context of the case study in terms of motivations for evolvability analysis, the description of evolvability subcharacteristics from the case perspective, and the experiences and observations we gained through the case study are also presented. Section 6 discusses the characterization of the qualitative and quantitative evolvability analysis methods, as well as validity evaluation. Section 7 reviews related work and Section 8 concludes the paper.

2. Software evolvability model

To improve the ability to understand and systematically analyze software architecture evolution, we introduced in our earlier work, a software evolvability model (Breivold et al., 2008b), which is used to provide a basis for analyzing and evaluating software evolvability. The model and its validation are based on the industrial requirements of a long-life software-intensive system within the automation domain. In this article, we refine the model and use it as the basis for the architecture evolvability analysis process (described in Section 3). Here we give a short overview of the model. The software evolvability model defines software evolvability as a multifaceted quality attribute (Rowe et al., 1994), and refines software evolvability into a collection of subcharacteristics that can be measured through a number of corresponding measuring attributes, as shown in Fig. 1. The idea with the model is to further derive the identified subcharacteristics until we are able to quantify them by defining metrics to determine relevant measuring attributes for each subcharacteristic, and/or make appropriate reasoning about the quality of service (QoS) for subcharacteristics that are difficult to quantify (e.g., architectural integrity, described below).

The identified evolvability subcharacteristics are based on a survey of the literatures (Breivold et al., 2011), an analysis of the software quality challenges and assessment (Fitzpatrick et al., 2004), the types of change stimuli and evolution (Chapin et al., 2001), and the taxonomy of software change based on various dimensions that characterize or influence the mechanisms of change (Buckley et al., 2005). In particular, they are the results from case studies (Breivold et al., 2008a,c), and are valid for a class of long life industrial software-intensive systems that are often exposed to many, in most cases evolutionary changes. For these types of systems we have identified the following subcharacteristics, with some examples of measuring attributes:

- **Analyzability** describes the capability of the software system to enable the identification of influenced parts due to change stimuli; its measuring attributes include modularity, complexity, and architectural documentation. Many measuring attributes can be domain-dependent.
- **Architectural integrity** describes the non-occurrence of improper alteration of architectural information; examples of measuring attributes include compatibility of deployment and communication patterns, resource allocations, programming styles, and inclusion of architectural documentation.
- **Changeability** describes the capability of the software system to enable a specified modification to be implemented and avoid unexpected effects; its measuring attributes include complexity, coupling, change impact, encapsulation, reuse, and modularity.
- **Extensibility** describes the capability of the software system to enable the implementations of extensions to expand or enhance the system with new features. It takes future growth into consideration. One might argue that extensibility is a subset of changeability. Due to the fact that about 55% of all change requests are new (Pigoski, 1996), we define extensibility explicitly as one subcharacteristic of evolvability. Examples of measuring attributes include scalability, resource constraints, and compliance to standards.
- **Portability** describes the capability of the software system to be transferred from one environment to another.
- **Testability** describes the capability of the software system to validate the modified software.
- **Domain-specific attributes** are the additional quality subcharacteristics that are required by specific domains. For example in a real-time systems domain, analysis of timing properties (response time, execution time, etc.) is important.

These evolvability subcharacteristics are the main enablers of evolvability. However, we do not exclude the possibility that other domains might have a slightly different set of subcharacteristics, in particularly with domain-specific attributes. For instance, the World Wide Web domain requires additional quality characteristics such as visibility, intelligibility, credibility, engagibility and differentiation (Fitzpatrick et al., 2004). Component exchangeability in the context of service reuse is another example within the distributed domain, e.g., wireless computing, component-based and service-oriented applications.

Software evolution is very often negatively influenced by architectural drift, feature creep, and progressive hardware dependence (Parnas, 1994). However, with the identified subcharacteristics in mind, we have a basis on which different systems can be examined in terms of evolvability. Any system design and architectural decisions that do not explicitly address one or more of these subcharacteristics will probably undermine the system's ability to be evolved. Therefore, the software evolvability model is a way to articulate subcharacteristics for an evolvable system that an architecture must support. It is established as a first step towards analyzing evolvability, a base and checkpoints for evolvability evaluation and improvement, and is an integral part of the qualitative and quantitative analysis of evolvability.

3. Software architecture evolvability analysis process

As evolvability and consequently the software evolvability model are complex with respect to the measurements and identification of subcharacteristics, the entire assessment process includes a set of complex procedures. For this reason we have identified a systematic assessment, the "software architecture evolvability analysis process" (AREA), with a goal to: (a) provide quality attribute subcharacteristics values, (b) identify the weak parts of the system architecture related to evolvability, and (c) analyze the quality attribute subcharacteristics of the possible evolutions of the system. The analysis can be regarded as a systematic technical review, and therefore can be carried out at many points during a system's life cycle, e.g., during the design phase to evaluate prospective candidate designs, validating the architecture before further commencement of development, or evaluating the architecture of a legacy system that is undergoing modification, extension, or other significant upgrades.

The evolvability analysis can be conducted by an internal assessment team or an external evaluation team. Having an internal assessment team requires discipline as it tends to be subject to more bias and influence, especially if the team is part of the organization that is responsible for evolving the architecture. An external assessment team is less affected by biased opinions, though its lack of knowledge concerning the system in focus is a weakness.

The results of the evolvability analysis process include: (i) the prioritized architectural requirements; (ii) stakeholders' evolvability concerns; (iii) candidate architectural solutions; and (iv) the impact of the architectural solutions on evolvability. It is a challenging task for an architect to choose between competing candidate architectural solutions and ensure that the system constructed from the architecture satisfies its stakeholders' needs. Therefore, the results from the evolvability analysis process are useful for an architect to design and evolve the architecture.

We introduced, in our earlier work, a qualitative assessment method (Breivold et al., 2008c) and a quantitative assessment method (Breivold and Crnkovic, 2010) for analyzing software evolvability at the architecture level, which we refine here in a common model with specifics in assessment activities (qualitative and quantitative). Note that "qualitative" and "quantitative" methods refer to the collection of information among the stakeholders, not the results of the provided subcharacteristics values. The overall AREA process is shown in Fig. 2.

The related artifacts in the evolvability analysis process include:

- *Change stimuli*: a stimulus is a change condition that needs to be considered from an architectural perspective. Change stimuli trigger an initiation of the architecture evolvability analysis process. A change stimulus can be a concrete change, a future change that we know will happen, or a change that we currently have no idea of, but belonging to a particular class of change related to environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software architecture evolution and embedded quality attributes; a change stimulus may result in a collection of potential requirements to which the software architecture needs to adapt.
- *Architectural concerns*: the IEEE 1471 standard (IEEE, 2000) defines architectural concerns as "interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns include system considerations such as performance, reliability, and evolvability". Here the concerns related to the evolvability should be provided – they are related to the

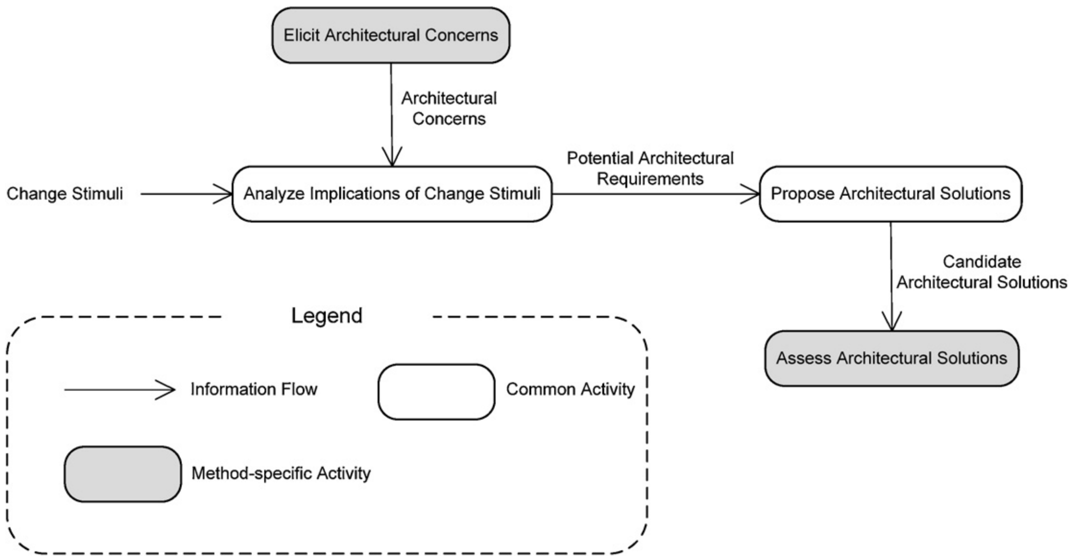

Elicit Architectural Concerns
Architectural Concerns
Change Stimuli
Analyze Implications of Change Stimuli
Potential Architectural Requirements
Propose Architectural Solutions
Candidate Architectural Solutions
Assess Architectural Solutions
Legend
Information Flow
Common Activity
Method-specific Activity

Fig. 2. Software architecture evolvability analysis process (AREA).

evolvability subcharacteristics and the domain-specific quality attributes.

- *Potential architectural requirements*: potential architectural requirements are requirements that influence software architecture and are essential for accommodating change stimuli.
- *Candidate architectural solutions*: candidate architectural solutions are potential solutions that reflect design decisions. The description of an architectural solution may include the following information:
  - *Problem description*: the problem and disadvantages of the original design of the architecture or fragment of the architecture.
  - *Requirements*: the new requirements that the architecture needs to fulfill.
  - *Improvement solution*: the architectural solution to design problems.
  - *Rationale and architectural consequences*: the rationale of the proposed solution and its architectural implications to evolvability.
  - *Estimated workload*: the estimated workload for implementation and verification.

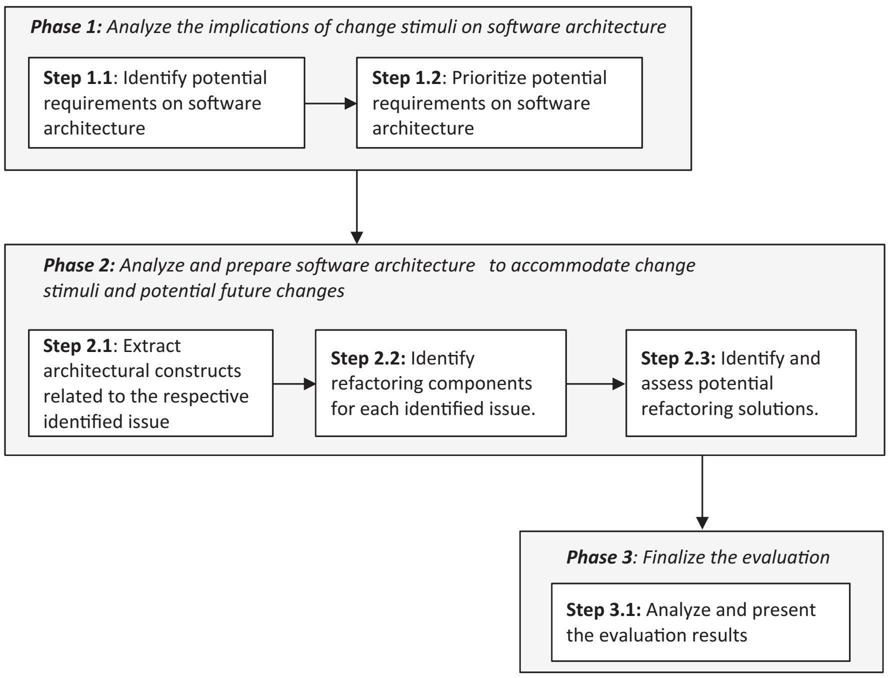
The main activities in the evolvability analysis process include:

- *Elicit architectural concerns*: this activity extracts architectural concerns with respect to evolvability subcharacteristics among stakeholders qualitatively or quantitatively.
  - *Qualitative elicitation*: architecture workshops are conducted so that the stakeholders discuss and identify potential architectural requirements against which the evolvability subcharacteristics are subsequently mapped. Thus the identified architectural requirements and their prioritization reflect stakeholders' architectural concerns with respect to evolvability subcharacteristics.
  - *Quantitative elicitation*: individual interviews with respective stakeholders are conducted so that stakeholders representing different roles provide their views and preferences of evolvability subcharacteristics through a pair-wise comparison of subcharacteristics with respect to their relative importance. Thus the weighting by preference of evolvability subcharacteristics from a stakeholder's perspective is quantified.
- *Analyze implications of change stimuli*: this activity analyzes the architecture for evolution and understands the impact of change stimuli on the current architecture. Accordingly, this activity focuses on defining the problems the architecture needs to solve, examining change stimuli and architectural concerns in order to obtain a set of potential architectural requirements.
- *Propose architectural solutions*: this activity proposes architecture solutions to accommodate a set of potential architectural requirements.
- *Assess architectural solutions*: this activity ensures that the architectural design decisions made are appropriate for software architecture evolution. The candidate architectural solutions are assessed against evolvability subcharacteristics, i.e., the implications of the potential architectural strategies and evolution path of the software architecture are assessed either qualitatively or quantitatively.
  - *Qualitative assessment*: the determination of potential architectural solutions is on a qualitative level in terms of their impact (positive or negative) on evolvability subcharacteristics.
  - *Quantitative assessment*: the judgment of how well each candidate architectural solution supports different evolvability subcharacteristics is quantified.

A typical evolvability assessment can be carried out in three half-day workshops, requiring the presence of architects, product manager, key software developers, and the person who conducts the assessment. The first workshop concentrates on the first two activities, i.e., "elicit architectural concerns" and "analyze implications of change stimuli". The second workshop focuses on identifying architectural solutions, and the third workshop focuses on the assessment of these architectural solutions. As we see from the general evolvability analysis process, the basic architecting activities such as analyzing implications of change stimuli and proposing architectural solutions are the same for both the qualitative and quantitative evolvability analysis. The major variation can be observed in the different details with respect to the elicitation and assessment of stakeholders' architectural concerns regarding the evolvability subcharacteristics of architectural solutions. The following subsections lay out the steps performed in the qualitative and quantitative evolvability analysis respectively.

3.1. Qualitative evolvability analysis method description

The qualitative evolvability analysis method starts with the identification of the implications of change stimuli, guides architects through the analysis of potential architectural requirements



Phase 1: *Analyze the implications of change stimuli on software architecture*
Step 1.1: Identify potential requirements on software architecture
Step 1.2: Prioritize potential requirements on software architecture
Phase 2: *Analyze and prepare software architecture to accommodate change stimuli and potential future changes*
Step 2.1: Extract architectural constructs related to the respective identified issue
Step 2.2: Identify refactoring components for each identified issue.
Step 2.3: Identify and assess potential refactoring solutions.
Phase 3*: Finalize the evaluation*
Step 3.1: Analyze and present the evaluation results

Fig. 3. The steps of the architecture evolvability analysis method.

that the software architecture needs to adapt to, and continues with identification of potential architecture refactoring solutions along with their implications. Through the analysis process, the implications of the potential improvement proposals and evolution path of the software architecture are analyzed with respect to evolvability subcharacteristics. The qualitative architecture evolvability analysis method, as shown in Fig. 3, is divided into three main phases.

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 considers both business and technical issues (Losavio et al., 2001), since the stimuli of changes come from both perspectives concerning, for example, environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software architecture and/or functionality. This phase includes two steps:

Step 1.1: *Identify requirements on the software architecture*. Any change stimulus may result in a collection of potential requirements that the software architecture needs to adapt to. The aim of this step is to extract requirements that are essential for software architecture enhancement so as to cost-effectively accommodate to change stimuli. This step is conducted in the form of workshops, where the stakeholders (e.g., product manager, and architects) discuss and identify architecture requirements.

Step 1.2: *Prioritize requirements on the software architecture*. In order to establish a basis for common understanding of the architecture requirements among stakeholders, the requirements identified from the previous step need to be prioritized.

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. It includes three steps.

Step 2.1: *Extract architectural constructs related to the respective identified issue*. In this step, we mainly focus on the identifications of architectural constructs (i.e., subsystems and components) that are related to each identified requirement.

Step 2.2: *Identify refactoring components for each identified issue*. 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. As part of this step, an assessment is made of the compatibility of the refactoring solutions and rationale with regard to design decisions made previously. The purpose of this assessment is to ensure architectural integrity.

Phase 3: Finalize the evaluation.

In this phase, the previous results are incorporated and structured into a collection of documents. This phase includes one step.

Step 3.1: *Present evaluation results*. The collected information from the evolvability assessment is summarized and presented to the stakeholders. This presentation can take the form of slides and might, in addition, be accompanied by a more complete written report. In this presentation, the person leading the evolvability evaluation recapitulates the steps of the assessment and all the information collected in the steps of the method, including: (i) the identified and prioritized requirements on the software architecture; (ii) the identified components/modules that need to be refactored for enhancement or adaptation; and (iii) refactoring investigation documentation which describes the current situation and solutions to each identified candidate component that need to be refactored, including estimated workload. These outputs are all uncovered, captured, and cataloged during the evaluation.

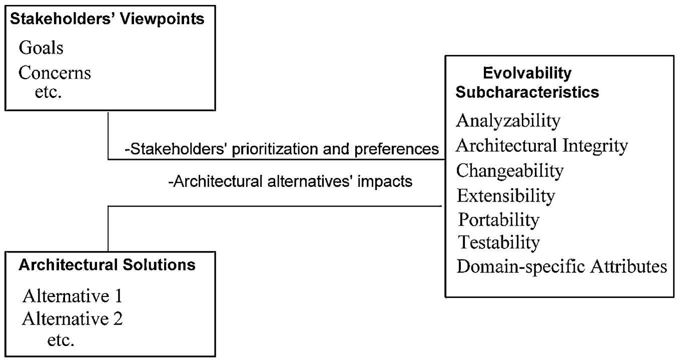


Fig. 4. Multiple-attribute decision making process.

3.2. Quantitative evolvability analysis method description

As architecture is influenced by stakeholders representing different concerns and goals, the business and technical decisions that articulate the architecture tend to exhibit tradeoffs and need to be negotiated and resolved. In circumstances when there is a lack of a shared view among stakeholders of prioritizations of evolvability subcharacteristics, to avoid the intuitive selection of architectural solutions, the quantitative evolvability analysis provides support to the decision making process and helps to avoid intuitive prioritization of evolvability subcharacteristics and intuitive choice of architectural solutions (Breivold and Crnkovic, 2010).

Our proposed approach focuses on two constituent steps of the qualitative evolvability analysis method in which tradeoff analysis is concerned, they are: step 1.2 – *Prioritize requirements on the software architecture* in phase 1, and step 2.3 – *Identify and assess potential refactoring solutions* in phase 2. These two steps entail subjective judgments with regard to preferences of evolvability subcharacteristics, as well as choice of architectural solutions. These subjective judgments constitute accordingly a multiple-attribute decision making process when architecting for evolvable software systems, as illustrated in Fig. 4. The stakeholders' preferences on evolvability subcharacteristics are determined by their different viewpoints, and the choice of architectural alternatives exhibits their respective impacts on evolvability. Moreover, the choice for an architectural solution is constrained by the stakeholders' preference information on evolvability subcharacteristics.

The quantitative evolvability assessment method is based on the Analytic Hierarchy Process (AHP) (Saaty, 1980), which is a multiple-attribute decision making method that enables quantification of subjective judgments. The qualitative method is extended with quantitative information that is needed for choosing among architectural solutions. The quantitative assessment method is divided into three main phases:

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

This phase elicits the architectural concerns among stakeholders and analyzes the architecture for evolution in order to accommodate change stimuli.

Step 1.1: *Elicit stakeholders' views on evolvability subcharacteristics*. In this step, individual interviews are conducted with respective stakeholders in order to elicit their views on evolvability subcharacteristics. Domain-specific attributes are identified as well. In addition, the interpretation of evolvability subcharacteristics in the specific context is discussed.

Step 1.2: *Extract stakeholders' prioritization and preferences of evolvability subcharacteristics*. In this step, stakeholders representing different roles provide their preferences of evolvability subcharacteristics through a pair-wise comparison of subcharacteristics (Q_i, Q_j) with respect to their relative importance. The AHP weighting scale shown in Table 1 is used to determine the relative importance of each evolvability subcharacteristic pair.

Table 1
Scale for pair-wise comparison.

Scale	Explanation
1	Variable *i* and *j* are of equal importance
3	Variable *i* is slightly more important than *j*
5	Variable *i* is highly more important than *j*
7	Variable *i* is very highly more important than *j*
9	Variable *i* is extremely more important than *j*
2, 4, 6, 8	Intermediate values for compromising between the numbers above

Note that the domain-specific attributes might comprise several additional quality characteristics that are required by a specific domain. Therefore, each of these domain-specific quality attributes is also included for pair-wise comparison together with the other evolvability subcharacteristics. The pair-wise comparison is conducted for all pairs, hence, $n(n-1)/2$ comparisons are made by each stakeholder. Afterwards, for each stakeholder, the AHP method is used to create a priority vector signifying the relative preference of evolvability subcharacteristics. As different stakeholder roles might have diversified preferences of evolvability subcharacteristics, for each evolvability subcharacteristic, we obtain a normalized preference by dividing the sum of the preference of each stakeholder role by the number of roles.

The description below lays out the calculation procedure, describing the calculation of preferences of subcharacteristics aggregated from stakeholders' perspectives. A matrix of pair-wise comparison is shown below, in which S_1 represents one stakeholder role, Q_1 and Q_2 and Q_k are evolvability subcharacteristics, and I_{ij} represents pair-wise comparison in terms of relative importance based on Table 1 (*note*: $I_{ij} = 1$ if $i = j$).

$$\begin{array}{c|cccc} S_1 & Q_1 & Q_2 & \ldots & Q_k \\ \hline Q_1 & I_{11} & I_{12} & \ldots & I_{1k} \\ Q_2 & I_{21} & I_{22} & & I_{2k} \\ \vdots & & & & \\ Q_k & I_{k1} & I_{k2} & & I_{kk} \end{array}$$

By applying AHP, we get the normalized preference weight information of subcharacteristic Q_i from the perspective of stakeholder S_1, as shown in Eq. (1):

$$PQ_{is1} = \frac{\sum_{j=1}^{k}(m_{ij})}{k} \quad (i \ \text{ is an integral and } \ 1 \le i \le k) \tag{1}$$

Likewise, the values indicating the preference weights of subcharacteristics ($Q_1, Q_2, \ldots, Q_k$) from the perspective of stakeholder S_2 are calculated. We designate them as $PQ_{1s2}, PQ_{2s2}, \ldots, PQ_{ks2}$. The same pattern applies to all the other stakeholder roles.

Given that the preference consistency is correct, the overall stakeholders' preference weight on subcharacteristic Q_i is calculated by aggregating the preferences from n number of stakeholders as shown in Eq. (2):

$$PQ_i = \frac{\sum_{j=1}^{n}(PQ_{isj})}{n} \quad (i \ \text{ is an integral and } \ 1 \le i \le n) \tag{2}$$

Phase 2: Analyze and prepare the software architecture to accommodate change stimuli.

This phase focuses on the identification of candidate architectural solutions to accommodate change stimuli.

Step 2.1: *Identify candidate architectural solutions*. In this step, candidate architectural solutions are identified along with their benefits and drawbacks.

Step 2.2: *Assess the impact of candidate architectural solutions on evolvability subcharacteristics*. In this step, system architects or the main technical responsible persons provide their judgment on how well each architectural alternative supports different evolvability subcharacteristics. This is done firstly through a pair-wise comparison of the architectural alternatives (Alt_i, Alt_j) with respect to a certain evolvability subcharacteristic, using the weighting scale in Table 1. Next, for each evolvability subcharacteristic, the AHP method is used to create a priority vector signifying the relative weight of how well different architectural alternatives support a specific evolvability subcharacteristic. Afterwards, recalling the overall weights, i.e., the stakeholders' preference weight of evolvability subcharacteristics and the weight of how well different architectural alternatives support a specific evolvability subcharacteristic, we can obtain a normalized value, designating the overall weight for each architectural alternative's support for evolvability in general.

The calculation procedure is carried out in a similar manner to the calculation of the subcharacteristics (see the description of phase 1 in the previous section) resulting in normalized support rates of the respective architectural alternative with respect to Q_1, in which $PAlt_{iq1}$ indicates the impact of the alternative Alt_i on subcharacteristic Q_1, i.e., how well each architectural alternative supports Q_1.

$$PAlt_{iq1} = \frac{\sum_{j=1}^{k} m_{ij}}{k} \quad (i \text{ is an integral and } 1 \le i \le k)$$

Likewise, the values indicating how well the alternatives support other subcharacteristics $(Q_2, \ldots, Q_k)$ are calculated following the same pattern.

Given that the judgment of the support of architectural alternatives for subcharacteristics is consistent, the overall weights of an alternative's support for evolvability is calculated by aggregating the preferences of subcharacteristics from the previous quantitative analysis (i.e., $PQ_1, PQ_2, \ldots, PQ_k$ in the previous subsection) as expressed in Eq. (3):

$$W_{Altm} = \sum_{i=1}^{k} (PQ_i \times PAlt_{mqi}) \quad (m \text{ is an integral and } 1 \le m \le k) \tag{3}$$

Phase 3: Finalize the evaluation.

In this phase, the previous results are incorporated and summarized.

Step 3.1: *Present evaluation results*. The evaluation results include (i) the identified evolvability subcharacteristics including domain-specific attributes; (ii) a quantified prioritization of evolvability subcharacteristics by respective stakeholders involved; (iii) a common understanding of the contexts of evolvability subcharacteristics; (iv) the architectural solution candidates identified as able to cope with change stimuli; and (v) a quantified prioritization of the impacts of each architectural candidate on evolvability subcharacteristics.

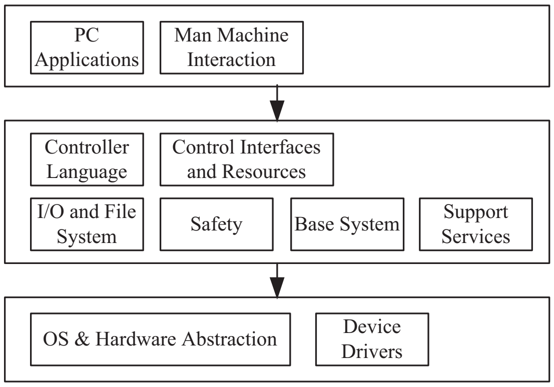


Fig. 5. A conceptual view of the original software architecture.

4. Case study I. Qualitative software evolvability analysis

This section describes the case study in which we applied the qualitative software evolvability analysis method. The system that we investigated is an automation control system at ABB.

4.1. *Context of the case study*

The case study was based on a large automation control system at ABB. During the long history of product development, several generations of automation controllers have been developed as well as a family of software products, ranging from programming tools to a variety of application software. The case study focused on the latest generation of the robot controller.

The robot 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 challenging to evolve. Our particular system is delivered as a single monolithic software package, which consists of various software applications developed by distributed development teams. These applications are aimed at specific tasks in painting, welding, gluing, machine tending, palletizing, etc. In order 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 was present in every product even though specific products did not require everything. As the system grew, it became more difficult to ensure that modifications to specific application software would not affect the quality of other parts of the software system.

The original coarse-grained architecture of the controller is depicted in Fig. 5. The lower layer provides an interface to the upper layer, and allows the source code of the upper layer to be compiled and used on different hardware platforms and operating systems. The complete set of interdependencies between subsystems within each layer is not captured in the figure.

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. To continue exploiting the substantial software investment made and to continuously improve the system for a longer productive lifespan, it became essential to explicitly address evolvability. We want to emphasize here that the problem raised is not a problem of maintainability. The major problems arose when brand new (very different) features, different development paradigms, or shifting business and organizational goals were introduced; therefore the problems were related to software evolvability.

4.2. Evolvability subcharacteristics from the case perspective

We provide the rationale for each evolvability subcharacteristic in conjunction with the case study context:

- *Analyzability*: the release frequency of the controller software was twice a year, with around 40 major new requirements that needed to be implemented with each release. These requirements may have an impact on different attributes of the system, and the possible impact must be analyzed effectively before the implementation of the requirements. Furthermore, analyzability includes the following requirements: the implemented changes should be easily isolated and tested; the resource utilization (memory and communication capacity) should be analyzable.
- *Architectural integrity*: a strategy for communicating architectural decisions that we discovered during this case study was to appoint members of the core architecture team as technical leaders in respective 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 the unconscious violation of architectural conformance. The requirements related to architectural integrity include documentation of architectural decision rationale, tool support for checking the deployment and communication patterns, and isolation of architectural layers.
- *Changeability*: due to the monolithic characteristic of the controller software, modifications in certain parts of the software package led to some ripple effects, and required the recompiling, reintegrating and retesting of the whole system. This resulted in inflexibility of patching, and customers had to wait for a new release even in the case of corrective maintenance and configuration changes. Requirements related to changeability include improved component cohesion using a standardized interface pattern, localization of functions, and the use of standard patterns for adding new services.
- *Portability*: the current controller software supports VxWorks and Microsoft Windows NT. In the meantime, there is also a need for openness in choosing between different operating system (OS) vendors, e.g., Linux and Windows CE, and possibly new OS's 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 the establishment of new market segments, it was decided that the controller must constantly raise its service level by supporting more functionality and providing more features, while keeping important non-functional properties.
- *Testability*: the controller software exposed a huge number of public interfaces which resulted in a tremendous amount of time spent on interface testing alone. Therefore, it was decided to reduce the number of public interfaces to around 10% of the original quantity. Besides that, due to the monolithic characteristic of the software, error corrections in one part of the software sometimes required retesting of the whole system. One decision taken was therefore to investigate the feasibility of testing only modified parts.
- *Domain-specific attributes*: the most important domain-specific attributes are related to real-time and potential problems with execution time. The critical real-time calculation demands of the controller software required reduced code size of the base software and runtime footprint.

4.3. Applying the qualitative evolvability analysis method

The main focus of our case study was to assess how well the architecture would support forthcoming requirements and understand their impact. The forthcoming requirements emerged due to the change stimuli brought about by the company's business strategy:

- Time-to-market requirements, such as building new products for dedicated market within short time.
- Increased ease and flexibility of the distributed development of diverse application variants.

The identification and analysis of the architectural requirements was performed by the core architecture assessment team which consisted of 6–7 people. It was a continuous maturation process from the first vision to concrete activities that will be described below. 2–3 people from the core architecture team identified the refactoring solution proposals for some components in the *Base System* subsystem. Workshops were conducted to discuss prioritization of architectural requirements and potential architectural solutions. The intention of the workshops is to reach a consensus among the stakeholders regarding the potential architectural requirements to focus on, as well as potential architectural solutions. The first author of the article worked within the architecture assessment core team, and proposed potential architectural solutions that would facilitate the implementation of the identified architectural requirements. These proposals were discussed with the main technical persons responsible and with the architects. The choice of architectural solutions was based on discussions with the system architects and prototyping through the whole architecture assessment process. All the architectural solutions were documented and transferred on to the implementation teams.

4.3.1. Phase 1 – step 1.1: identify requirements of the software architecture

Due to the change stimuli mentioned earlier, the main requirements of the software architecture and the refined activities for each requirement were proposed by the core assessment team, and are listed below:

R1. Modular architecture

- 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.
- Investigate dependencies between the existing extensions.

R2. Reduced architecture complexity

- Define system interfaces between subsystems and reduce the number of public interface calls.
- Add support for real-time task isolation management.
- Introduce a new scripting language to improve support for application development, since some modern scripting languages are flexible, productive and reduce the need to recompile.

R3. Enable distributed development of extensions with minimum dependency

Table 2
Mapping between evolvability subcharacteristics and architecture requirements.

Subcharacteristics	Requirements
Analyzability	R1. Modular architecture R2. Reduced architecture complexity
Changeability	R1. Modular architecture R2. Reduced architecture complexity
Extensibility	R3. Enable distributed development of extensions with minimum dependency
Portability	R4. Portability
Testability	R2. Reduced architecture complexity R5. Impact on product development process
Domain-specific attribute	R6. Minimized software code size and runtime footprint

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

R4. Portability

- Investigate portability across target operating system platforms.
- Investigate portability across hardware platforms.

R5. Impact on the product development process

- Investigate the implications of restructuring the automation controller software, with respect to product integration, verification and testing.

R6. Minimized software code size and runtime footprint

- Investigate enabling mechanisms, e.g., properly partitioning functionality.

These requirements were then checked against the evolvability subcharacteristics to justify whether the realization of each requirement would lead to an improvement of the subcharacteristics (or possibly a deterioration, which would then require a tradeoff decision). Table 2 summarizes how the identified architectural requirements are related to the evolvability subcharacteristics.

It may be noted that architectural integrity is omitted from this table. This is because, in the case study, architectural integrity was handled by documenting the architectural choices for handling potential architectural requirements, and the rationales for the choice of architectural solutions along with their impacts on evolvability subcharacteristics. This will be detailed later.

4.3.2. Phase 1 – step 1.2: prioritize requirements on the software architecture

With the consideration of not disrupting ongoing development projects, the criteria for requirement prioritization were: (i) enable the building of existing types of extensions after refactoring and architecture restructuring; (ii) enable new extensions, and simplify interfaces that are difficult to understand and/or may have negative effects on implementing new extensions. Based on these criteria, R1–R3 were prioritized architectural requirements.

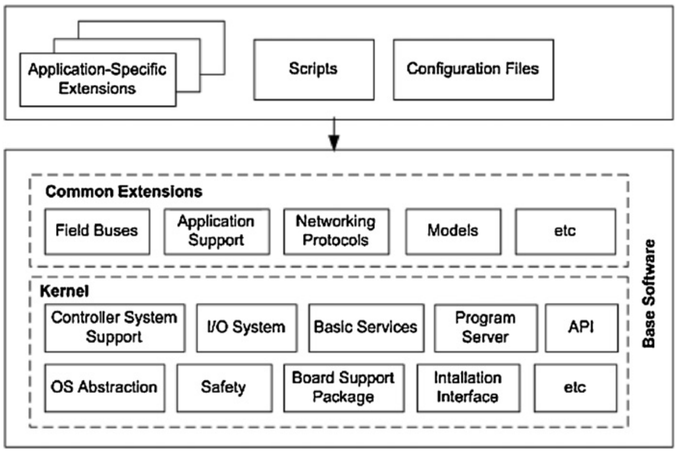


Fig. 6. A revised conceptual view of the software architecture.

4.3.3. Phase 2 – step 2.1: extract architectural constructs related to the respective identified issues

We demonstrate the use of the method by exemplifying with R3 (enable distributed development of extensions with minimum dependency). To enable distributed application development, there is a need to 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. Accordingly, we extracted architectural constructs that were related to the realization of distributed development. Details on how we go further with the extracted architectural constructs are described below in steps 2.2 and 2.3.

4.3.4. Phase 2 – step 2.2: identify refactoring components for each identified issue

To enable distributed development of extensions with minimum dependency, the strategy of separate concerns was applied to isolate the effect of changes to parts of the system (Breivold et al., 2008c), i.e., separate the general system functions from the hardware, and separate application-specific functions from generic and basic functions. Based on the extracted cross-cutting concerns, the refactoring was conducted by merging subsystems/components, re-grouping of components, breaking down components and re-structuring them into new subsystems. Thus, the original architecture shown in Fig. 5 was proposed to be changed to the architecture shown in Fig. 6. Consequently, some subsystems and components need to be adapted and reorganized to enable the architecture restructuring. For instance, the *PC Applications* and *Man Machine Interaction* in the original architecture become *Application-specific Extensions*, whereas the *OS & Hardware Abstraction* in the original architecture becomes a subsystem in the kernel in the new architecture. We also identified a collection of components that needed refactoring. Some of them were the components within the low-level *basic services* subsystem for resource allocations, e.g., the *semaphore ID management* component, and the *memory allocation management* component. These components needed to be adapted because functionality needed to be separated from resource management, in order to achieve the build- and development-independency between the kernel and extensions.

4.3.5. Phase 2 – step 2.3: identify and assess potential refactoring solutions from technical and business perspectives

The complete assessment of components cannot be presented due to space limitations and company confidentiality. Therefore, we select a subset, and exemplify with one component example that needed to be refactored. We will focus on the technical perspective and discuss in terms of the following views: (i) problem

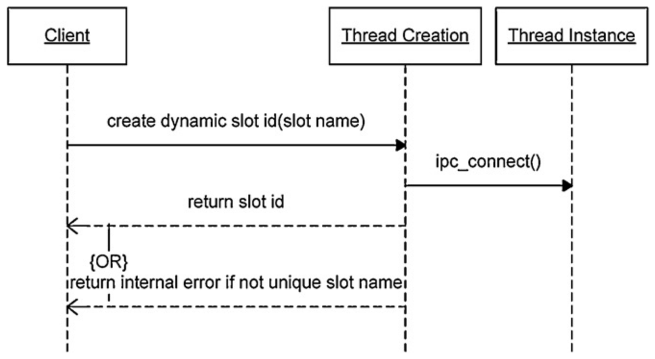

Client
Thread Creation
Thread Instance
create dynamic slot id(slot name)
ipc_connect()
return slot id
{OR}
return internal error if not unique slot name

Fig. 7. The inter-communication component after refactoring.

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; and (iv) architectural consequences: the architectural implications of the deployment of the component on evolvability subcharacteristics.

Component example: inter-process communication

This component belongs to the *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 are specified in the startup command file for thread creation. There is no dynamic allocation of connection slot.

Requirements: the refactoring of this component is directly 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 also be available in the kernel, to facilitate movement 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 Fig. 7. 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.

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 enabled. 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 dynamically requires resources, which are limited due to real-time requirements. This may require additional analysis including a trade-off analysis of possible solutions.

4.3.6. Phase 3 – step 3.1: present evaluation results

Until this step, key architectural requirements were identified; components that needed to be refactored were identified; the stakeholders established a common understanding of potential improvement strategies and the evolution path for the software architecture. In Table 3, we summarize the implications of the refactored component example on evolvability subcharacteristics.

Table 3
Implications of the component examples on evolvability subcharacteristics (+, positive impact; −, negative impact).

	IPC component refactoring
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

4.4. Qualitative evolvability analysis: lessons learned

In the qualitative evolvability analysis method, the architecture tradeoff analysis is reflected in two constituent steps: (i) during architecture workshops, the stakeholders prioritize potential architectural requirements, which are mapped against evolvability subcharacteristics. By prioritizing the potential architectural requirements based on pre-defined criteria, evolvability subcharacteristics are implicitly prioritized by stakeholders; (ii) after the workshop, the identified architectural choices are qualitatively analyzed with respect to their impacts and support for evolvability subcharacteristics. Therefore, we see two aspects which we can further explore and make more explicit.

1. Explicit stakeholders' views on prioritization and preferences of evolvability subcharacteristics.
   Rationale: depending on their roles in the development and evolution of a software system, the stakeholders usually have different concerns, i.e., interests which pertain to the system's development, its operation or evolution. Consequently, architecting for an evolvable software system implies that an architect needs to balance numerous stakeholders' concerns that are reflected in their prioritization and preferences of evolvability subcharacteristics. When the prioritization and preferences of evolvability subcharacteristics are not explicitly expressed by involved stakeholders, it becomes difficult to determine the dimensions along which a system is expected to evolve.
   Related activities performed in the qualitative evolvability analysis method: this aspect was treated implicitly in the step Prioritize requirements in the first phase, in which the potential architectural requirements were mapped against evolvability subcharacteristics, and were then prioritized based on predefined criteria. As a result, the choice of prioritized architectural requirements implicitly sets priority ranking on evolvability subcharacteristics.
2. Quantification of the impact of architectural solution alternatives on evolvability subcharacteristics.
   Rationale: choosing an architectural solution that satisfies evolvability requirements is vital to the evolution and success of a software system. Nonetheless, each solution candidate is associated with multiple attributes, as the choice of any solution alternatives may probably cause varied tradeoffs among evolvability subcharacteristics. Hence, it is important

to understand how an architectural alternative supports different evolvability subcharacteristics, especially when there are several alternatives to choose from, each of which exhibits varied support for evolvability subcharacteristics. Consequently, these alternatives need to be ranked, and at the same time, reflect stakeholders' preference information on evolvability subcharacteristics.

Related activities performed in the qualitative evolvability analysis method: the determination of potential architectural solutions along with their impact on evolvability subcharacteristics was qualitatively handled in the step Identify and assess refactoring solutions in the second phase, by examining the rationale of a solution proposal along with its architectural implications (positive or negative impact) of the deployment of the component on evolvability subcharacteristics.

5. Case study II. Quantitative software evolvability analysis

This section describes the case study in which we applied the quantitative software evolvability analysis method. The system that we investigated is a mobile network node software architecture at Ericsson.

5.1. Context of the case study

The case study was based on an assessment of the mobile network architecture with respect to the evolvability of a logical node at Ericsson. The main purpose of the logical node is to handle control signaling for and keep track of user equipment such as mobiles using a certain type of radio access. This is a mature system that was introduced about ten years ago and has been refined since then. The system is expected to remain on the market for years to come, and thus, needs to be easy to maintain and evolve.

The system software[1] is divided into two levels: (i) the platform level which consists of operating systems, a distributed processing environment and application support; and (ii) the application level, which comprises of a control system and a transmission system. The control system is designed to process high-level protocols and control user traffic data flow in the transmission system. The transmission system is responsible for transport, routing and processing of user traffic.

The case study focused on one of the challenges that the system needs to meet, i.e., In-Service Software Upgrade (ISSU). The system downtime is divided into planned and unplanned downtime. Planned downtime is imposed by maintenance routines, such as correction package loading. Unplanned downtime is imposed by automatic recovery mechanisms in the system and manual restarts of the system due to a system failure. The actual downtime for a network is largely dependent on the frequency of the planned downtime events. There are two scenarios connected with planned downtime.

- *Update of a release*: the corrections to a release include either correction packages that are distributed to all customers or single corrections that are made for specific customers only, and may be later included in the correction packages. These corrections are planned patches, and can be updated at runtime. During the update of a release, no configuration data needs to be changed or updated.
- *Upgrade of a release*: a release is upgraded to a new release with changed characteristics of the network node, e.g., changes in node configuration parameters, major changes in software and hardware. At present, this causes downtime of the node. During an upgrade, when the new software has been installed, the node is restarted (automatically or manually), and the local configuration that a customer maintains is converted to a new format if needed.

[1] For reasons of confidentiality, no more details about the system are presented here.

A main driver of the design and evolution of the system is the achievement of non-stop operation with minimum service impact. Therefore the focus of our study is the second scenario which is the main cause of node downtime, because the node restart, being part of each upgrade, causes service interruption for 5–10 min. The architecture must support this emerging requirement of In-Service Software Upgrade in order to evolve. The evolvability analysis in the case study focused on understanding the impact on the current architecture and investigating its potential evolution path, taking the emerging software upgrade requirement into consideration.

5.2. Evolvability subcharacteristics from the case perspective

We describe below each evolvability subcharacteristic in conjunction with the case study context.

Analyzability: the release frequency of the system in the case study is twice a year, with various new customer requirements, strategic functionality and characteristics implemented in each release. In addition, the software development organization is feature-oriented, i.e., software developers are not grouped based on subsystems; instead, they are grouped to implement a certain feature, and therefore often need to work across various subsystems. This requires that the software system needs to be easily understood and have the capability to be analyzed in terms of the impact on the software caused by the introduction of a change. From the ISSU perspective, it was decided that an ISSU solution should be easy to understand for the development organization.

Architectural integrity: in the development of the network and node system, several architectural design patterns, guidelines as well as design rules with respect to conformity, modeling and style guides have been articulated in an architectural specification document. All these fundamental principles (strategies and guidelines) govern the design and evolution of the system, and therefore are clearly defined and communicated. In addition to the strategies that guide software developers in order to fulfill requirements (features of direct value for a user), system strategies are also defined to fulfill non-functional attributes of high priority. From the ISSU perspective, to enable ISSU implementation, it was decided that ISSU rules should be followed. It was also decided that it was necessary to check whether a potential ISSU architectural design has any violations against these general design rules. If any ISSU component must break the rules, it is essential to record the rationale for such design decision and strategy.

Changeability: from the ISSU perspective, four aspects were concerned: (i) how well can other architectural changes fit into the ISSU solution; (ii) many kinds of application changes shall be possible without special upgrade code, e.g., backward-compatible interfaces; (iii) it shall be as easy as possible to write special upgrade code if needed, and (iv) how easy is it to change the ISSU solution itself once it is used?

Extensibility: the system must constantly raise the level of service by extending existing features or adding new ones. From the ISSU perspective, one concern was to identify if there were any limitations when introducing the ISSU solution.

Portability: the current node software supports VxWorks and Linux on a number of hardware variants. In the future, a possible scenario could be to change the operating system or support new hardware.

Testability: the system has a number of variants based on the selection of the hardware configuration and the capacity level of the node. Therefore, an important concern is the ease of testing and debugging parts of the system individually, and extracting test data from the system. From the ISSU perspective, three aspects were concerned: (i) would ISSU influence the number of variants? (ii) would it be possible to conduct component tests? and (iii) would it be possible to reproduce test cases?

Domain-specific attributes: two domain-specific attributes were identified:

- *Capacity* is an attribute that describes the subscriber and throughput capacity with various radio access types. It depends on the traffic pattern and dimensioning of the operator network. A logical node is dimensioned for a specific load capacity. The admission control functions and limits given by capacity licenses would limit the number of subscribers allowed to enter the node and the number of resources occupied by these subscribers. Besides this, overload protection mechanisms are implemented in case of internal failure within a node, network failure, reconfiguration or incorrect node dimensioning. From the ISSU perspective, three aspects were concerned: (i) ISSU total time; (ii) capacity impact during ISSU, and (iii) capacity impact during normal execution.
- *Availability* is an attribute that describes the ability to keep the node in service, i.e., to keep the downtime to a minimum. It is also called In-Service Performance (ISP) by the domain experts that we interviewed. The system needs to be tolerant against both hardware- and software-related failures so that the services provided by the node are always available. The recovery functions aim to provide a non-stop mode of operation of the system, i.e., to recover from both software and hardware failures with minimal inconvenience to the attached subscribers. From the ISSU perspective, three aspects were concerned: (i) redundancy of critical components during ISSU; (ii) impact of ISSU solution's complexity, and (iii) impact of software or hardware failures during upgrade.

5.3. *Applying the quantitative evolvability analysis method*

The main focus in our case study was to identify, with respect to the system function In-Service Software Upgrade (ISSU), which among a set of architecture candidates has the most potential for fulfilling the quality requirements of the system.

5.3.1. *Phase 1 – step 1.1: elicit stakeholders' views on evolvability subcharacteristics*

The change stimuli to the evolution of the node architecture in the case study came from the ever-growing stringent requirement for "In-Service-Performance". Based on the identified change stimuli, the main architectural requirements were defined in order to evaluate potential architectural solution alternatives:

- The atomic component for which an upgrade is performed must have backward compatible interfaces during the upgrade.
- The old configuration data (including node-internal replicate data) format must be available during the whole upgrade.
- The replicated subscriber data format must be available in its old format until the upgrade is finished.
- It must be known on which software version each atomic component executes.
- There must be a component which controls the upgrade and is aware of the progress.

To elicit stakeholders' views on evolvability subcharacteristics, we performed interviews with key personnel and software designers to understand architectural challenges over the years in general, as well as the challenges that the architecture is facing due to various emerging requirements, e.g., distributed development, and increased productivity required due to including more features in each product release. In addition, we interviewed the following relevant stakeholders to elicit their views on evolvability subcharacteristics:

- Three system architects.
- Two software designers involved in the logical node's development.
- The system owner.

These stakeholders possess a wide range of expertise, covering platform development, communication protocol, node configuration, monitoring and upgrade. The evolvability analysis methodology was presented to the stakeholders being interviewed in order to give them a clear idea of the entire process, and the value of their contribution. The interviews were conducted separately for each stakeholder with the intension that his/her preference judgment should not be influenced by other people. The interviews were semi-structured, and the interviewees were free to discuss their main concerns about evolvability subcharacteristics from their perspective. We also extracted the stakeholders' view on important domain-specific attributes (which were identified during the interviews), i.e., capacity and availability.

5.3.2. *Phase 1 – step 1.2: extract stakeholders' prioritization and preferences of evolvability subcharacteristics*

We extracted the information on the stakeholders' preferences after we had gone through the list of evolvability subcharacteristics, and clarified the definition of each subcharacteristic in the stakeholders' specific context. This was to ensure that each stakeholder's prioritization of subcharacteristics was built upon the same ground. We asked each stakeholder to provide us with preferences of evolvability subcharacteristics from his/her own perspective. Table 4 shows a system architect's preferences of evolvability subcharacteristics.

The other system architects' preferences on evolvability subcharacteristics were collected and calculated in the same manner. Table 5 summarizes all the system architects' preferences on evolvability subcharacteristics, along with their aggregated prioritizations based on Eq. (2), as described in Section 3.2.

After the process of extracting system architects' preferences on evolvability subcharacteristics, it was interesting to note that the three system architects shared almost the same view of prioritization of subcharacteristics. They had a shared order of prioritization (starting from high to low priority) – availability, capacity, testability, changeability, extensibility, and analyzability. This is a good indication of the alignment of preferences among architects.

In the same way, we also gathered quality preferences for the other stakeholder roles, and realized that different stakeholder roles have different preferences of evolvability subcharacteristics. A summary of different stakeholder preferences is presented in Table 6.

The reason why we aggregate preferences per stakeholder role is that each role represents its respective viewpoint and needs, and thus, we assume that the primary preference differentiation lies between the different stakeholder roles. During the process of extracting stakeholders' views on evolvability subcharacteristics, we also performed a consistency check for each stakeholder's comparisons based on AHP (Saaty, 1980). Table 7 summarizes the consistency ratio scores for each stakeholder.

The research in Saaty (1980) suggested that if the consistency ratio is smaller than 0.10, a participant' comparisons are consistent enough to be useful, and the AHP method can yield meaningful results. It is also pointed out in Saaty (1980) that, in practice, higher

Table 4
Preferences of evolvability subcharacteristics provided by a software architect.

	Analyzability	Integrity	Changeability	Extensibility	Portability	Testability	Availability	Capacity
Analyzability	1	3	1/3	1/3	5	1	1/4	1/3
Integrity	1/3	1	1/6	1/6	2	1/3	1/8	1/7
Changeability	3	6	1	2	5	1/3	1/2	1/2
Extensibility	3	6	1/2	1	3	1/3	1/2	1/2
Portability	1/5	1/2	1/5	1/3	1	1/7	1/9	1/8
Testability	1	3	3	3	7	1	1/3	1/2
Availability	4	8	2	2	9	3	1	1
Capacity	3	7	2	2	8	2	1	1

Table 5
Aggregated subcharacteristics.

Architects	Analyzability	Integrity	Changeability	Extensibility	Portability	Testability	Availability	Capacity
Architect A	0.077	0.030	0.135	0.110	0.023	0.158	0.249	0.219
Architect B	0.059	0.047	0.084	0.064	0.057	0.105	0.407	0.176
Architect C	0.082	0.036	0.096	0.096	0.038	0.123	0.309	0.220
Aggregated	0.073	0.038	0.105	0.090	0.039	0.128	0.322	0.205

Table 6
Preferences of evolvability subcharacteristics provided by respective stakeholder roles.

Stakeholders	Analyzability	Integrity	Changeability	Extensibility	Portability	Testability	Availability	Capacity
Architects	0.073	0.038	0.105	0.090	0.039	0.128	0.322	0.205
Designers	0.105	0.125	0.103	0.108	0.042	0.154	0.322	0.041
System owner	0.061	0.189	0.111	0.108	0.023	0.112	0.350	0.046
Aggregated	0.080	0.117	0.106	0.102	0.035	0.131	0.331	0.098

Table 7
Consistency ratios for stakeholders.

Stakeholders	Architect A	Architect B	Architect C	Designers	System owner
Consistency ratio	0.061	0.109	0.039	0.088	0.046

values are often obtained, which indicates that 0.10 may be too low a limit. But it is an indication of the approximate value of the expected consistency ratio. As we see from Table 7, only architect B's value (0.109) is slightly more than 0.10. However, the value is still acceptable considering that 0.10 is a tough limit for the degree of consistency. Consequently, all the data we obtained from the stakeholders are trustworthy. The aggregated values of all the involved stakeholder roles, as shown in Table 6, indicate that availability has the highest priority, followed by testability, architectural integrity, changeability, extensibility, capacity, analyzability, and portability.

5.3.3. *Phase 2 – step 2.1: identify candidate architectural solutions*

Two architectural alternatives were developed for the In-Service Software Upgrade requirement in our study. For reasons of confidentiality we cannot give full descriptions of the candidate architectural solutions, but the architectural alternatives describe two variations of how to handle execution resource management. Two types of computing resource (processors) management are used to fulfill the capacity and In-Service Performance (ISP) requirements: (i) application processors that are optimized for node control and traffic control logic, and (ii) device processors that are optimized for communication/protocol logic to handle time-critical traffic data flow and control signaling termination. Specifically, the two candidate architectural solutions are:

- *Alt*1 – *slot by slot concept*: the overall idea is to take one board after another out of service for upgrade to a new release. During the In-Service Software Upgrade, the boards running with old software will coexist and interact with the boards running with new software.
- *Alt*2 – *zone concept*: the overall idea is to divide the node into two zones, i.e., in one zone, all components run old software, and in the other zone, components run new software.

Both solutions have their respective benefits and drawbacks. For instance, the *slot by slot concept* has the benefit of having board redundancy under control during ISSU and that the existing mechanisms in the architecture facilitate the potential implementations of ISSU. On the other hand, the *zone concept* has the benefit that backward compatibility is not needed for application and device processors. But both solutions face several drawbacks such as time required for ISSU, interface changes, and others. Therefore, it was not an easy task to directly decide which alternative would be more optimal than the other.

5.3.4. *Phase 2 – step 2.2: assess the impact of candidate architectural solutions on evolvability subcharacteristics*

To assess the impact of ISSU candidate architectural solutions on evolvability subcharacteristics, we actively cooperated with the system architects at Ericsson. The two candidate architectural solutions were rated with respect to how well they support each evolvability subcharacteristic. This information was provided by the three system architects because they possess the whole system perspective and technical knowledge. The values indicating the support weights of the two alternatives with respect to evolvability subcharacteristics are summarized in Table 8.

Table 8
Prioritization of the two architectural alternatives.

	*Alt*1 – slot by slot	*Alt*2 – zone
Analyzability	0.667	0.333
Integrity	0.500	0.500
Changeability	0.250	0.750
Extensibility	0.333	0.667
Portability	0.500	0.500
Testability	0.333	0.667
Availability	0.750	0.250
Capacity	0.800	0.200

5.3.5. Phase 3 – step 3.1: present evaluation results

Until this step, key domain-specific attributes and candidate architectural solutions were identified; stakeholders' preferences of evolvability subcharacteristics as well as each candidate solution's support of evolvability subcharacteristics were quantified.

Consequently, considering the prioritization weights of evolvability subcharacteristics in Table 6, together with the values indicating each alternative's support of evolvability subcharacteristics shown in Table 8, the overall weight for *Alt*1 is calculated based on Eq. (3) as:

$$\begin{aligned} W_{Alt1} &= 0.080 \times 0.667 + 0.117 \times 0.500 + 0.106 \times 0.250 \\ &\quad + 0.102 \times 0.333 + 0.035 \times 0.500 + 0.131 \times 0.333 \\ &\quad + 0.331 \times 0.750 + 0.098 \times 0.800 = 0.560 \end{aligned}$$

Similarly, $W_{Alt2} = 0.440$, which indicates that, *Alt*1 *slot by slot concept* is the preferred solution supporting evolvability.

5.4. Quantitative evolvability analysis: lessons learned

This section summarizes our main experiences and lessons learned through applying the quantitative analysis method during the case study.

5.4.1. Experiences

By applying the quantitative analysis method, we have improved the ability to explicitly extract stakeholders' views on evolvability subcharacteristics and the quantification of the support of candidate architectural solutions for evolvability. Thus, intuitive choices of architectural solutions are avoided during software evolution. We list below two tangible benefits that were perceived and reported by the organization's stakeholders who were involved in the study.

Quantification of stakeholders' preferences of evolvability subcharacteristics: in this case study, different stakeholder roles had different concerns relative to the software system. For instance, the software designers mentioned three main aspects that were considered important from their perspective, i.e., functionality, ease of understanding, and source code level performance; whereas the system owner focused on domain-specific attributes (availability and capacity), functionality, and time-to-market/time-to-customer. These concerns are critical from specific stakeholders' perspectives, and will thus influence how they prioritize evolvability subcharacteristics. According to the stakeholders we interviewed, thinking in terms of "subcharacteristics", was not new for them. But previously they had not been able to quantify the importance of the various -abilities for their system. The quantitative evolvability analysis method provided a structured way to extract and quantify the opinions of the stakeholders of various roles who are involved in the software architecture decision process through individual discussions and interviews. In addition, the quantification results served as a communication vehicle for discussions of development concerns among various stakeholders when individual preferences were quantitatively identified and highlighted.

Quantification of the impact of architectural alternatives on evolvability: in this case study, recalling the stakeholders' preference weight of evolvability subcharacteristics and the weight of how well different alternatives support a specific evolvability subcharacteristic, we obtained a normalized value, designating the overall weight for each alternative's support for evolvability, and indicating which was the preferred candidate architectural solution. In addition, we also interviewed the system architects after the execution of the method in the form of a discussion meeting to collect their opinions on whether the method had produced relevant results. According to them, these results can definitely serve as a basis for further discussions on the choice of architectural solution. Most importantly, the systematic analysis approach, including documentation of the reasoning in each step, was most valuable, as it provided them an active countermeasure against arbitrarily making some design decisions that were otherwise often based on intuition because of personal experience and available expertise.

5.4.2. Lessons learned

In the case study, we conducted a series of informal interviews with the stakeholders that participated in the evolvability analysis. During the interviews, we asked questions that were designed to extract and clarify the stakeholders' perception of evolvability subcharacteristics. In this process, cost was not explicitly considered. Cost involves development cost, maintenance and evolution cost, and concerns time-to-market. In the case study, we put cost into consideration when candidate architectural solutions had been identified as it became more concrete to estimate the workload for each solution. On the other hand, in order to carry out software evolution efficiently, the cost aspect could also be considered upfront and explicitly evaluated together with the evolvability subcharacteristics.

6. Discussion

In this section, we summarize the characteristics of the qualitative and quantitative analysis methods, and discuss validity evaluation.

6.1. Characterization of the two methods

Both qualitative and quantitative methods can be used as an integral part of the software development and evolution process to assess software architectures for evolution. They share the common themes of (i) systematically addressing quality requirements driven by change stimuli, and (ii) assisting architects in analyzing the impact of potential architectural solutions on evolvability subcharacteristics before determining the potential evolution path for the software architecture. There are also variations between the two methods as detailed below and summarized in Table 9.

Application contexts:

- *Stakeholders' perception of quality attributes.* Software architecture is influenced by system stakeholders (Bass et al., 2003). In circumstances when there are numerous stakeholder roles, representing different and sometimes contradictory concerns and goals, an explicit quantitative assessment of stakeholders' preferences of evolvability subcharacteristics will strengthen qualitative data, and assist architects in making architectural design decisions, especially when there is not a clear view within an organization on important quality attributes and their prioritization.

Table 9
Characterizations of the qualitative and quantitative evolvability analysis methods.

	Application contexts	Approaches used	Analysis output
Qualitative evolvability analysis	Common perception of important quality attributes and their prioritization within organization	Architecture workshops with all involved stakeholders to discuss prioritization of potential architectural requirements	Identified and prioritized potential architectural requirements
	A preferred architectural solution can be decided based on the qualitative impact data	Architecture workshops with architects to discuss architectural solutions and qualitative impacts on evolvability	Qualitative analysis of impact of architectural solutions on evolvability subcharacteristics
Quantitative evolvability analysis	Numerous stakeholder roles representing different concerns	Interviews with individual stakeholders to discuss preferences of evolvability subcharacteristics	Quantified stakeholders' preferences of evolvability subcharacteristics to make stakeholders' evolvability concerns explicit
	Unclear perception and prioritization of important quality attributes.	Architecture workshops with architects to discuss architectural solutions and quantitative impacts on evolvability.	Quantified prioritization of impact of candidate architectural solutions on evolvability
	Difficult to decide the preferred architectural solution based on qualitative data	Analytic Hierarchical Process method	

- *Impact of candidate architectural solutions on evolvability*. Architects must often make architectural design decisions and give preference to a certain architectural solution. In circumstances when there are multiple architectural alternatives to choose among, each of which exhibits divergent impacts on evolvability subcharacteristics, a quantitative assessment of the impact of candidate architectural solutions on evolvability subcharacteristics will guide and support architects in avoiding intuitive decisions in software architecture evolution, especially when the qualitative data is not sufficient for determining a preferred candidate architectural solution.

Approaches used in the analysis process:

The qualitative evolvability analysis is mainly conducted through (i) architecture workshops in which all involved stakeholders participate to identify and prioritize potential architectural requirements, and (ii) architecture workshops in which the architects discuss potential architectural solutions along with their qualitative impacts on evolvability. The quantitative evolvability analysis is based on AHP (Saaty, 1980) and conducted through (i) interviews with respective stakeholder to extract individual stakeholder's preference of evolvability subcharacteristics; and (ii) architecture workshops in which the architects discuss potential architectural solutions along with their quantitative impacts on evolvability.

Analysis output:

The main output of the qualitative evolvability analysis method includes the identified and prioritized potential architectural requirements, identified components that need to be refactored, candidate architectural solutions along with their qualitative evolvability impact analysis data, as well as test scenarios. The main output of the quantitative evolvability analysis method includes quantified prioritization of evolvability subcharacteristics among stakeholders, and identified candidate architectural solutions along with their quantitative evolvability impact data.

The evolvability assessment methods explicitly address evolvability subcharacteristics and analyze potential architectural solutions' impact on these subcharacteristics. These assessments result in suggested architectural solutions and a potential evolution path of the software architecture. Some quality attributes can be quantified, and they affect the evolution of a system. Based on our experiences in industrial settings, the domain-specific attributes address a lot of these quality attributes, such as performance, capacity and availability in the second case study. As the evolvability analysis is done before the actual architecture transformation, it is difficult to measure/quantify the quality attributes (e.g., performance) of the architecture before it is actually implemented. Therefore, these quality attributes are quantified using AHP, based on subjective judgments of how well the potential architecture choice would support the specific quality attributes.

The choice of which analysis method to use is based on the specific application contexts and the expected analysis output. The following questions are related to application contexts, and can be used as checkpoints (answered with 'Yes' or 'No') for determining when to use which analysis method:

- Are there numerous stakeholder roles with divergent concerns and goals? (Y/N).
- Are the important quality attributes that concern the evolution of the system in focus clear within the software development organization? (Y/N).
- Is it difficult to determine a preferred candidate architectural solution among the multiple architectural alternatives due to their various impacts on evolvability subcharacteristics? (Y/N).

Fig. 8 illustrates the decision diagram for choosing the appropriate analysis method based the answers to the questions. The first two checkpoints concern the stakeholders' perception of quality attributes, and are related to the first phase of both qualitative and quantitative analysis. The third checkpoint concerns selecting a preferred architectural solution, and relates to the second phase of the two methods. It is therefore possible to combine the qualitative and quantitative analysis methods, e.g., starting with a qualitative analysis and complement with quantitative data by using AHP, or vice versa. Depending on the answers to the questions, the corresponding phase of either the qualitative or quantitative analysis can be selected.

6.2. Validity

Our software architecture evolution research is based on empirical studies. The formulation of the evolvability model was originally built upon our observations and experiences of working with many different types of industrial systems from different domains, several workshop discussions (Breivold and Crnkovic, 2010; Breivold et al., 2008b,c), and the involvement of practitioners in the discussions. To further confirm the proposed evolvability subcharacteristics, we performed a systematic review of software

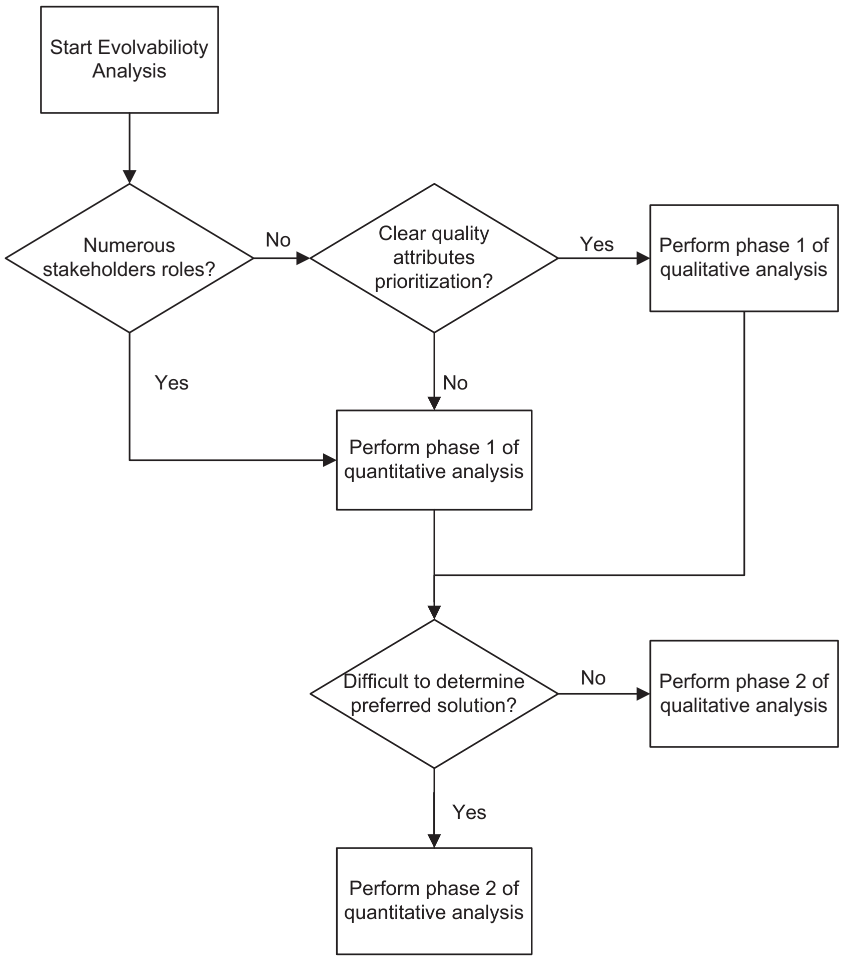

Start Evolvabilioty Analysis
Numerous stakeholders roles?
No
Clear quality attributes prioritization?
Yes
Perform phase 1 of qualitative analysis
Yes
No
Perform phase 1 of quantitative analysis
Difficult to determine preferred solution?
No
Perform phase 2 of qualitative analysis
Yes
Perform phase 2 of quantitative analysis

Fig. 8. A decision diagram for choosing between the qualitative and quantitative methods.

architecture evolution research (Breivold et al., 2011). This systematic literature review was based on a formalized and repeatable process to document relevant knowledge on architecting for evolvability. We searched in seven scientific databases that provide important and high-impact full text journals and conference proceedings, covering software evolution related areas with a broad range of topics, such as software quality models, process models, software quality metrics, and software architecture evaluation. To further ensure the strength of inferences, we defined quality criteria that indicate the credibility of an individual study. Based on a pre-defined search strategy and a multi-step selection process, we identified, from the initial 3349 papers, 82 primary studies for further analysis of quality attributes related to evolvability. The citation status and publication sources, along with their impact factor also represent the high quality level of the studied literature. Moreover, the model and the analysis methods were applied in two large-scale industrial software systems in different domains.

These case studies are in essence based on action research (Argyris et al., 1985), i.e., the researchers participate in the process and perform empirical observations. We discusses below the validity evaluations of performing the cases based on Yin (2002).

Conclusion validity: Conclusion validity (Yin, 2002) is concerned with the relationship between the treatment and the outcome. In the *qualitative* evolvability analysis, conclusion validity is addressed through: (i) architecture workshops with stakeholders to extract potential architectural requirements; (ii) the involvement of software architects and senior software developers in the analysis process to discuss the impact of candidate architectural solutions on evolvability, and (iii) the researchers' experiences and involvement in the software product development.

In the *quantitative* evolvability analysis, as the answers to how important the evolvability subcharacteristics relate to each other is in the form of a subjective judgment, the answers tend not to be exactly the same for all participants, especially among stakeholders representing different roles. This was noticed in the case study, in which the preferences among stakeholder roles differed whereas the architects had a shared preference view on evolvability subcharacteristics. We saw this as a positive indication that there was an organizational alignment among architects. On the other hand, even the same participant might not provide exactly the same answer in terms of pair-wise comparison weights should the study be repeated. Therefore, the interviews were centered on asking a series of questions that were open-ended, i.e., conversational responses, to gain information about respective stakeholder's view and interpretations of evolvability subcharacteristics. This was to ensure that the stakeholders had a well-elaborated and clarified understanding of evolvability subcharacteristics in their specific domain context before providing meaningful pair-wise comparison weights for evolvability subcharacteristics and impacts of architectural alternatives on evolvability. Moreover, the calculation of consistency ratio in the AHP method also helped to check the consistency level of the individuals' answers.

The first author took part in the evolvability analysis in both cases. All experiences are thus first-hand. The evolvability assessments followed the structured process, which reduced the influence of the authors' participation. In addition, other participants in the cases provided us with material to make the conclusions (e.g., lessons learned, and experiences) less subjective. The risk of bias has been further decreased through the involvement of other researchers (co-authors) in the analysis of the experiences.

Internal validity: Internal validity (Yin, 2002) concerns the connection between the observed behavior and the proposed explanation for the behavior. In the quantitative evolvability analysis case, during the process of extracting stakeholders' preferences of evolvability subcharacteristics, a remark from the software designers was that a designer may have different valuation of evolvability subcharacteristics depending on the different subsystems that he/she has previously worked with. This is because different subsystems may have different quality attribute requirements in focus. Although we encouraged them to try to think at the system level, it may still become a threat to the study when extracting software designers' preferences of evolvability subcharacteristics. However, in the qualitative analysis process, this threat was addressed through the architecture workshops in which all the stakeholders could discuss their perception and prioritization of architectural requirements, and thus, could reach a consensus.

Construct validity: Construct validity (Yin, 2002) is concerned with the relation between theory and observation. In both the qualitative and quantitative analysis cases, we informed the participants about the evolvability analysis process so that they became aware of the purpose and the intended results of the studies. Furthermore, we set the context and expectations for the remainder of the evaluation activities. One threat that exists, however, is in the qualitative analysis case, during the architecture workshops, when all the stakeholders discuss potential architectural requirements and their prioritization. Some people might not tell their true opinions if they deviate from those of the others. But, this type of threat was addressed in the quantitative analysis process in which separate interviews were conducted individually with respective stakeholders.

External validity: External validity (Yin, 2002) is concerned with generalization. In both the qualitative and quantitative cases, the participants represented the different roles of stakeholders that are involved in software development. Therefore, there is no threat in the selection of participants. In addition, based on our experiences in both case studies, although the systems belong to different domains – automation and telecommunication domains, the analysis methods seemed to be generally applicable. However, one threat to external validity is that there are some similarities between the two cases, such as large, complex, long-lived software-intensive systems with strong requirements for backward compatibility and no evolution breaks. Another threat is that both companies are large international ones located in Sweden, and thus might impose some social and cultural behavior on people, especially during interviews and workshops.

7. Related work

A comprehensive description of the related work can be found in our systematic review of software architecture evolution research (Breivold et al., 2011). The discussion of related work in this section concerns three topics: quality models, qualitative architecture analysis, and quantitative architecture analysis. Their common characteristics are that they either do not consider evolvability, or they do not provide any method to assess evolvability, and can only be partially used in the evolvability assessment process.

7.1. Quality models

In quality models, quality attributes are decomposed into various factors, leading to various quality factor hierarchies. Some well-known quality models are McCall et al. (1977), Dromey (1996), Boehm et al. (1978), ISO 9126 and FURPS (Grady and Caswell, 1987). In summary, none of these existing quality models is dedicated to evolvability analysis. Certain evolvability subcharacteristics are disregarded or not explicitly addressed in these models. Although we can enrich respective quality model through integrating the missing elements, it is still difficult to extend/adapt each quality model for the purpose of software evolvability analysis for two reasons: (i) the existing quality models are intended to evaluate the quality of software in general; (ii) most of the quality models are more driven towards the final coded software product, and do not explicitly take into account the analysis and design stage (Losavio et al., 2001), which is an essential part in the proposed evolvability assessment process.

7.2. Qualitative architecture analysis

There are many approaches to how to perform qualitative architecture analysis. For instance, the *attribute-based architectural style* (ABAS) (Klein et al., 1999) associates architectural styles with reasoning frameworks that are based on quality-attribute-specific models for particular quality attributes. With this approach, in order to determine potential evolution paths of an architecture, the preferences and tradeoffs among evolvability subcharacteristics must be considered.

Another example is the *lightweight sanity check for implemented architectures* (LiSCIA) method (Bouwers and van Deursen, 2010), that focuses on maintainability and reveals potential problems as a software system evolves. The limitations of LiSCIA are: (i) it depends heavily on the evaluator's opinion; (ii) it only aims to discover potential risks related to maintainability; and (iii) the use of only a single viewpoint (module view-type) sets a limit on covering all potential risks.

The *knowledge-based assessment approach* (Del Rosso and Maccari, 2007) evaluates the evolution path of software architecture during its lifecycle based on the knowledge of the stakeholders involved in the software development organizations. The outcomes of the assessment are current architecture overview, main issues found, and optionally, recommendations for their resolutions. Although this approach addresses evolvability, there is no description of the authors' perception of evolvability, and a lack of explicit consideration of the multifaceted feature of software evolvability.

7.3. Quantitative architecture analysis

There are several quantitative analysis methods and a few that are representative for related work are *Architecture Level Modifiability Analysis* (ALMA) (Bengtsson et al., 2004), *Decision support method* (Svahnberg, 2004) and *Cost Benefit Analysis Method* (CBAM) (Kazman et al., 2006). For a deeper description see our systematic review (Breivold et al., 2011).

Architecture Level Modifiability Analysis (ALMA) (Bengtsson et al., 2004) analyzes modifiability based on scenarios that capture future events a system needs to adapt to in its lifecycle. Depending on the goal of the analysis, the output from an ALMA evaluation varies between: (i) maintenance prediction to estimate required effort for system modification to accommodate future changes; (ii) architecture comparison for optimal candidate architecture, and (iii) risk assessment to expose the boundaries of software architecture by explicitly considering environment and using complex change scenarios that the system shows inability to adapt to. ALMA does not cover all the evolvability subcharacteristics, which is a shortcoming that we address with our work.

Decision support method (Svahnberg, 2004) quantitatively measures stakeholders' views on the benefits and liabilities of software architecture candidates and relevant quality attributes. The method is used to understand and choose optimal candidate architecture from among software architecture alternatives. Although the primary data collection is comprised of subjective judgments, influenced by the knowledge, experiences and opinions of stakeholders, the data collection of stakeholders' subjective opinions is quantifiable. Thus, any disagreements between the participating stakeholders can be highlighted for further discussions.

One way to quantitatively analyze architecture is to address economic valuation perspective and estimate the required effort for system modification to accommodate future changes. *Cost Benefit Analysis Method* (CBAM) (Kazman et al., 2006) is an architecture-centric economic modeling approach that can address long-term benefits of a change along with its implications on the complete product lifecycle. This method quantifies design decisions in terms of cost and benefits analysis, and prioritizes changes to architecture based on perceived difficulty and utility. Compared with these approaches, the economic evaluation is not one of the major outputs from our evolvability analysis methods, in which we only make estimates of the required implementation workload of the architectural solution candidates.

8. Conclusions

Motivated by the need to understand software architecture evolution and to investigate ways to analyze software evolvability to support this evolution, the central theme of this paper focuses on two particular aspects: (i) identify software characteristics that are necessary to constitute an evolvable software system, and (ii) assess evolvability in a systematic manner. We have proposed and described the software architecture evolvability analysis process (AREA), which provides several repeatable techniques for supporting software architecture evolution:

- *Software evolvability model* refines evolvability into a collection of subcharacteristics, and is established as a first step towards analyzing and quantifying evolvability; This model provides a basis for analyzing and evaluating software evolvability, and a check point for evolvability evaluation and improvement.
- *Qualitative evolvability analysis method* focuses on improving the ability to systematically understand and analyze the impact of change stimuli on software architecture evolution.
- *Quantitative evolvability analysis method* provides quantifications of stakeholders' evolvability concerns and the impact of potential architectural solutions on evolvability.

These techniques have been applied in two industrial projects driven by the need to improve software evolvability. Based on our experiences, both the qualitative and quantitative analysis methods can be used as an integral part of the software development and evolution process. Throughout the process of evolvability analysis at ABB, the architecture requirements and the rationale of the choice of an architectural solution for architecture transition became more explicit, and better founded and documented. The analysis results were well accepted by the stakeholders involved in the analysis process, and became a blueprint for further implementation improvement. Throughout the process of evolvability analysis at Ericsson, the importance of various quality attributes perceived among different stakeholders was quantified and became more explicit. This quantification also served as a communication vehicle for further discussions among stakeholders. In both cases, by analyzing architectural improvement proposals with respect to their implications on evolvability subcharacteristics, we further avoided an ad hoc choice of potential evolution paths of software architecture. Our plans are to further complement the quantitative analysis method with a cost aspect to better support design decisions, and validate on additional, independent cases.

Acknowledgements

We would like to acknowledge Swedsoft (http://www.swedsoft.se/) for providing the forum for cooperation and exchange of ideas between companies, and Ericsson for making it possible to conduct our case study. In particular, we would like to thank Anders Fredén, Magnus Norlander, Erik Lind, and all the others involved in the case study at Ericsson for their help and participation in our case study, and for their valuable input and positive feedback.

References

Argyris, C., Putnam, R., Smith, D.M., 1985. Action Science: Concepts, Methods, and Skills for Research and Intervention. Jossey–Bass Social and Behavioral Science, ISBN 0875896650.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison-Wesley Professional, ISBN 0321154959.

Bengtsson, P., Lassing, N., Bosch, J., van Vliet, H., 2004. Architecture-level modifiability analysis (ALMA). Journal of Systems and Software 69, 129–147.

Bennett, K., Rajlich, V.,2000. Software maintenance and evolution: a roadmap. In: Conference on the Future of Software Engineering. ACM.

Bennett, K., 1996. Software evolution: past, present and future. Information and Software Technology 38, 673–680.

Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., Merritt, M.J., 1978. Characteristics of Software Quality. North-Holland, ISBN 0444851054.

Borne, I., Demeyer, S., Galal, G.H., 1999. Object-oriented architectural evolution. Lecture Notes in Computer Science 1743, 57–64.

Bouwers, E., van Deursen, A., 2010. A lightweight sanity check for implemented architectures. IEEE Software 27, 44–50.

Breivold, H.P., Crnkovic, I., Land, R., Larsson, S.,2008a. Using dependency model to support architecture evolution. In: 4th International ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE.

Breivold, H.P., Crnkovic, I., 2010. An extended quantitative analysis approach for architecting evolvable software systems. In: Computing Professionals Conference Workshop on Industrial Software Evolution and Maintenance Processes (WISEMP).

Breivold, H.P., Crnkovic, I., Eriksson, P.J., 2008b. Analyzing software evolvability. In: 32nd IEEE International Computer Software and Applications Conference (COMPSAC).

Breivold, H.P., Crnkovic, I., Land, R., Larsson, M., 2008c. Analyzing software evolvability of an industrial automation control system: a case study. In: 3rd International Conference on Software Engineering Advances (ICSEA), pp. 205–213.

Breivold, H.P., Crnkovic, I., Larsson, M., 2011. A systematic review of software architecture evolution research. Journal of Information and Software Technology, http://dx.doi.org/10.1016/j.infsof.2011.06.002.

Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G., 2005. Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice 17, 309–332.

Cai, Y., Huynh, S., 2007. An evolution model for software modularity assessment. In: Fifth International Workshop on Software Quality.

Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., Tan, W.G., 2001. Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice 13, 3–30.

Clements, P., Kazman, R., Klein, M., 2002. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, ISBN 0-201-70482-X.

Del Rosso, C., Maccari, A., 2007. Assessing the architectonics of large, software-intensive systems using a knowledge-based approach. In: The Working IEEE/IFIP Conference on Software Architecture (WICSA).

Dromey, R.G., 1996. Cornering the chimera. IEEE Software 13, 33–43.

Fitzpatrick, R., Smith, P., O'Shea, B., 2004. Software quality challenges. In: The Second Workshop on Software Quality at the 26th International Conference on Software Engineering.

Garlan, D., 2000. Software architecture: a roadmap. In: Conference of the Future of Software Engineering.

Godfrey, M.W., German, D.M., 2008. The past, present, and future of software evolution. Frontiers of Software Maintenance (FoSM), 129–138.

Grady, R.B., Caswell, D.L., 1987. Software Metrics: Establishing a Company-wide Program. Prentice-Hall, ISBN 0-13-821844-7.

IEEE-1471, 2000. IEEE Recommended Practices for Architectural Description of Software-Intensive Systems.

ISO/IEC 9126-1, International Standard, Software Engineering. Product Quality – Part 1: Quality Model.

Kazman, R., Asundi, J., Klein, M., 2006. Quantifying the costs and benefits of architectural decisions. In: 23rd International Conference on Software Engineering (ICSE).

Klein, M.H., Kazman, R., Bass, L., Carriere, J., Barbacci, M., Lipson, H., 1999. Attribute-based architecture styles. In: First Working IFIP Conference on Software Architecture (WICSA).

Land, R., Crnkovic, I., 2007. Software systems in-house integration: architecture, process practices, and strategy selection. Information and Software Technology 49, 419–444.

Larsson, S., Wall, A., Wallin, P., 2007. Assessing the influence on processes when evolving the software architecture. In: International Workshop on Principles of Software Evolution.

Lehman, M.M., Ramil, J.F., Kahen, G., 2000. Evolution as a noun and evolution as a verb. In: Workshop on Software and Organization Co-evolution.

Losavio, F., Chirinos, L., Perez, M.A., 2001. Quality Models to Design Software Architectures. Technology of Object-Oriented Languages and Systems.

Madhavji, N.H., Fernandez-Ramil, J., Perry, D., 2006. Software Evolution and Feedback: Theory and Practice. John Wiley & Sons, ISBN 0-470-87180-6.

McCall, J.A., Richards, P.K., Walters, G.F., 1977. Factors in Software Quality. National Technical Information Service (NTIS).

Medvidovic, N., Taylor, R.N., Rosenblum, D.S., 1998. An architecture-based approach to software evolution. In: International Workshop on the Principles of Software Evolution.

Mens, T., Guéhéneuc, Y.G., Fernández-Ramil, J., D'Hondt, M., 2010. Guest editors' introduction: software Evolution. IEEE Software 0740-7459/10.

Mens, T., Magee, J., Rumpe, B., 2010b. Evolving software architecture descriptions of critical systems. Computer 43, 42–48.

Nehaniv, C.L., Wernick, P., 2007. Introduction to software evolvability. In: International IEEE Workshop on Software Evolvability.

Parnas, D.L., 1994. Software aging. In: 16th International Conference on Software Engineering.

Pigoski, T.M., 1996. Practical Software Maintenance. John Wiley & Sons Inc, ISBN-13: 978-0471170013.

Rowe, D., Leaney, J., 1997. Evaluating evolvability of computer based systems architectures – an ontological approach. In: International Conference and Workshop on Engineering of Computer-based Systems.

Rowe, D., Leaney, J., Lowe, D., 1994. Defining systems evolvability – a taxonomy of change. In: IEEE Conference and Workshop on Engineering of Computer-based Systems.

Saaty, T.L., 1980. The Analytical Hierarchy Process. McGraw-Hill, New York.

Svahnberg, M., 2004. An industrial study on building consensus around software architectures and quality attributes. Information and Software Technology 46, 805–818.

Weiderman, N.H., Bergey, J.K., Smith, D.B., Tilley, S.R., 1997. Approaches to Legacy System Evolution. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-97-TR-014.

Yin, R.K., 2002. Case Study Research: Design and Methods. Sage Publications Inc, ISBN-10: 0761925538.

Yu, L., Ramaswamy, S., Bush, J., 2008. Symbiosis and software evolvability. IT Professional 10, 56–62.

Dr. Hongyu Pei Breivold is a principle scientist within the industrial software architecture technical area at ABB Corporate Research, Västerås, Sweden. Her experience includes participation of company-wide technology and software-intensive system development projects within different domains. She received a Ph.D. in Computer Science & Engineering from Mälardalen University, Sweden in 2011. Her main research interests include software architecture, architecture analysis and evaluation, and software evolution in general.

Prof. Ivica Crnkovic received the Ph.D. Degree ('91) in computer science, and before that the M.Sc. ('81) in computer science and M.Sc. in theoretical physics ('84) all from the University of Zagreb, Croatia. After 15 years of work in industry, he moved to academia 1999. He is a professor of software engineering at Mälardalen University, Sweden, and a professor at Faculty of Electrical Engineering in Osijek, Croatia. He is a co-author of three books and a co-editor several books and proceedings, and a co-author of more than 150 refereed publications on software engineering topics. He was a general chair of CompArch 2011 conference federated event, ESEC/FSE 2007 conference, CBSE2006, Euromicro SEAA 2005, and he organized several other conferences and workshops in the area of software engineering. During 2009–2011 he has chaired the Comparch steering committee, and he is a co-chair of the Euromicro SEAA technical committee, and a member of the steering committee of ESEC. His research interests include component-based software engineering, software architecture, software configuration management, software development environments and tools, and software engineering in general.

Dr. Magnus Larsson is a research manager at ABB Corporate Research where ongoing research is in the area of software architecture and user experience. Magnus sees the importance of doing a thorough work on the software architecture to reach great user experience. Magnus is also engaged part time in the academia as an adjunct professor at Mälardalen University in the area of component-based software engineering and software architecture. Since 2007 Magnus has been invited to the Swedish foundation for strategic research, to do work on new research programs, levering Magnus combined experience from industry and academia.