Assessing Optimal Software Architecture Maintainability

Jan Bosch
University of Groningen
Department of Computing Science
PO Box 800,
NL9700AV Groningen, Netherlands
e-mail: Jan.Bosch@cs.rug.nl
www: http://www.cs.rug.nl/~bosch

PerOlof Bengtsson
Blekinge Institute of Technology
Department of Software Engineering
and Computer Science
S-372 25 Ronneby, Sweden
e-mail: PerOlof.Bengtsson@ipd.hk-r.se
www: http://www.ipd.hk-r.se/pob

Abstract

Over the last decade, several authors have studied the maintainability of software architectures. In particular, the assessment of maintainability has received attention. However, even when one has a quantitative assessment of the maintainability of a software architecture, one still does not have any indication of the optimality of the software architecture with respect to this quality attribute. Typically, the software architect is supposed to judge the assessment result based on his or her personal experience. In this paper, we propose a technique for analysing the optimal maintainability of a software architecture based on a specified scenario profile. This technique allows software architects to analyse the maintainability of their software architecture with respect to the optimal maintainability. The technique is illustrated and evaluated using industrial cases.

1. Introduction

Over the last decade, we have come to the understanding that the challenging task of the development of software systems no longer is to provide the required functionality, but rather to fulfil the quality requirements of the system. Quality requirements, e.g. performance and maintainability, typically require explicit attention during development in order to achieve the required levels. Traditionally, software engineers have performed their designs based on assumed relations between design solutions and quality requirements. Based on these relations, software design is often performed as an implicit, relatively ad-hoc process. Unfortunately, in our experience the assumptions of software engineers are not always correct, e.g. [10].

A second trend one can identify over the last decade is the increasing appreciation of the importance of the architecture of a software system. The software architecture defines the most fundamental design decisions and the consequent decomposition of the system into its main components and relations between these components. Within the community, we have become to understand the fact that the quality attributes of a software system are, to a large extent, constrained by its architecture. Consequently, the quality requirements most central to the success of the software system should drive the design of the software architecture. Unfortunately, the design of software architectures is often performed in a similar fashion as software design as a whole, i.e. as an experience-driven, implicit activity.

One of the main causes for the aforementioned problems is the fact that software engineers have few techniques available for predicting the quality attributes of a software system before the system itself is available. Typically, we *measure* the quality attributes, once the system has been put in operation. The difficulty to predict quality attributes is particularly problematic at the software architecture design level. During this phase, the first design decisions are taken that are fundamental to the system, but we have little means of evaluating these decisions until much later in the development process.

Of course, this problem has not been ignored within the software architecture community and several examples of software architecture analysis techniques exist. Typically, we can categorize these techniques into *architecture oriented* and *quality attribute oriented* techniques. Architecture oriented analysis techniques can be further decomposed into architect-based evaluation, such as architecture assessment teams [2], and stakeholder-based assessment, e.g. SAAM [11]. Quality attribute oriented techniques can be decomposed into qualitative, typically comparing two architectures for a quality attribute, and quantitative analysis techniques, e.g. predicting real-time behaviour [1] or our work on maintainability assessment [3]. Typically, architecture-oriented assessment is performed once the software architecture phase is finished, whereas quality attribute assessment is often performed as part of the iterative architecture design process.

Assuming the availability of quantitative assessments of the driving quality attributes of a software architecture, the software architect will typically wonder "how good is this?". In the case of maintainability, the quality attribute we focus on in this paper, this is often one of the core questions during design. Maintainability is defined as the ease with which we may perform maintenance tasks of different kinds. In practise maintainability is often considered as related to the cost, or effort, it takes to perform the necessary maintenance tasks. For instance, assume the result of the software architecture maintainability assessment being that the maintenance effort will be equivalent to 10 full-time maintenance engineers per year, one would like to know whether an alternative architecture would allow for a substantially lower maintenance cost.

The contribution of this paper is a solution to the problem outlined above. We present a technique that, based on a maintenance scenario profile and an impact analysis of this profile on an architecture, calculates what the theoretically minimal maintenance effort would be for a software system based on the available maintenance scenario profile. Our technique is based on the results of the scenario-based maintainability assessment which controls the different types of maintenance considered in the assessment. In its current form the scenario impact analysis id only concerned with non-corrective maintenance, i.e. the maintenance tasks that are not concerned with correcting bugs.

The remainder of this paper is organized as follows. In the next section, assessment and design of software architectures is discussed. Section 3 discusses the quantitative assessment of maintainability using change scenarios and impact analysis. The technique for the assessment of the optimal maintainability of a software system is presented in section 4, whereas the subsequent section exemplifies the technique using an industrial case. Finally, related work is discussed in section 6 and the paper is concluded in section 7.

2. Software Architecture Assessment and Design

When discussing software architecture assessment and design, one has to start with defining software architecture. One frequently used definition is "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them" [2]. Some authors, e.g.[14], consider the software architecture to consist of the structure, in terms of components and their relations, but also the design constraints and rules, as well their rationale.

The architecture of a software system constrains its quality attributes, as discussed in the introduction. This, in combination with the fact that the earliest design decisions are the hardest to revoke, indicates the important of an explicit and objective software architecture design process. In [6], we present a method for software architecture design that employs explicit assessment of and design for the quality requirements of a software system. Although this method is one instance, we believe that this method is rather prototypical for general software architecture design. Therefore, we describe the method briefly in this section to understand the role of quantitative maintainability assessment in the context of software architecture design.

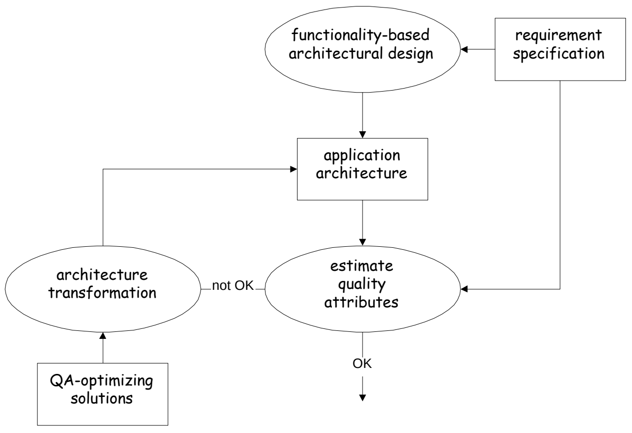
functionality-based architectural design
requirement specification
application architecture
architecture transformation
not OK
estimate quality attributes
OK
QA-optimizing solutions

Figure 1. Software architecture design method

The architecture design process, as shown in figure 1, can be viewed as a function taking a requirement specification that is taken as an input to the method and an architectural design that is generated as output. However, this function is not an automated process and considerable effort and creativity from the involved software architects is required. The software architecture is used for the subsequent phases, i.e. detailed design, implementation and evolution. The design process starts with a design of the software architecture based on the functional requirements specified in the requirement specification. Although software engineers generally will not design a system less reliable or maintainable, the quality requirements are not explicitly addressed at this stage. The result is a first version of the application architecture design. This design is evaluated with respect to the quality requirements. Each quality attribute is given an estimate in using a qualitative or quantitative assessment technique. The estimated quality attribute values are compared to the values in the requirements specification. If all estimations are as good or better than required, the architectural design process is finished. Otherwise, the third stage is entered: architecture transformation. During this stage, the architecture is improved by selecting appropriate quality attribute-optimizing transformations. Each set of transformations (one or more) results in a new version of the architectural design. This design is again evaluated and the same process is repeated, if necessary, until all quality requirements are fulfilled or until the software engineer decides that no feasible solution exists. In the latter case, the software architect needs to renegotiate the requirements with the customer. The transformations, i.e. quality attribute optimizing solutions, generally improve one or some quality attributes while they affect others negatively.

In order to be able to determine whether the software architecture design process is finished and, if not, to be able to select transformations, assessment techniques are required provide quantitative data concerning the relevant quality attributes of the system. In the next section, we discuss a technique for assessing maintainability of software architectures.

3. Assessing Maintainability

Quality requirements such as performance and maintainability are, in our experience, generally specified rather weakly in industrial requirement specifications. In some of our cooperation projects with industry, e.g. [4], the initial requirement specification contained statements such as “The maintainability of the system should be as good as possible” and “The performance should be satisfactory for an average user”. Such subjective statements, although well intended, are useless for the evaluation of software architectures.

When one intends to treat the architecture of a software system explicit in order be able to predict the quality attributes of the system early in the design process, one is required to specify quality requirements in sufficient detail. One common characteristic for quality requirements is that stating a required level without an associated context is meaningless. For instance, also statements such as “Performance = 20 transaction per second” or “The system should be easy to maintain” are meaningless for architecture evaluation.

One common denominator of most quality attribute specification techniques is that some form of *scenario profile* is used as part of the specification. A scenario profile is a set of scenarios, generally with some relative importance associated with each scenario. The profile used most often in object-oriented software development is the *usage profile*, i.e. a set of usage scenarios that describe typical uses for the system. The usage profile can be used as the basis for specifying a number of, primarily operational, quality requirements, such as performance and reliability. However, for other quality attributes, other profiles can be used. For example, for specifying safety we have used *hazard scenarios* and for maintainability we have used *change scenarios*.

3.1.Scenario Profiles

There are two ways of specifying profiles for quality attributes, i.e. complete and the selected profiles. When defining a complete profile for a quality attribute, the software engineer defines all relevant scenarios as part of the profile. For example, a usage profile for a relatively small system may include all possible scenarios for using the system. Based on this complete scenario set, the software engineer is able to perform an analysis of the architecture for the studied quality attribute that, in a way, is complete since all possible cases are included.

The alternative to complete profiles are selected profiles. Selected profiles are analogous to the random selection of sample sets from populations in statistical experiments. Assuming a large population of possible scenarios, e.g. change scenarios, one selects individual scenarios that are made part of the sample set. A complete sample set is what we, so far, have referred to as a scenario profile. Assuming the selection of scenarios has been done careful, one can assume that the profile represents an accurate representation of the scenario population. Consequently, results from architecture assessment based on the profile, will provide accurate results for the system as a whole and not just for the profile. Obviously, the actual selection of scenarios is not random because these are created by a software architect. Therefore, we have to spend explicit effort on making sure that the scenario profile is as representative as possible. In [5], we present three approaches to defining profiles, evaluate these using an experiment and identify one of the three as the preferred technique for scenario profile specification.

3.2.Maintainability Assessment

A software system is developed according to a requirement specification, i.e. $RS = \{r_1, \ldots, r_p\}$ where r_i is a atomic functional requirement or a quality requirement. Based on this requirement specification, the software architect or architecture team designs a software architecture consisting of a set of components and relations between these components, i.e. $SA = \{C, R\}$. The set of components is $C = \{c_1, \ldots, c_q\}$ and each component $c_i = \{I, cs, rt\}$ has an interface I, a code size cs and a requirement trace rt. The set of relations is $R = \{r_1, \ldots, r_r\}$, where each relation is

$r_i = \{c_{source}, c_{dest}, type\}$. Thus, relations are directed and one-to-one.

To assess the maintainability of a software architecture, we make use of a maintenance profile consisting of a set of change scenarios, $MP = \{cs_1, \ldots, cs_s\}$, where each change scenario defines a set of changed and added requirements, $cs_i = \{r_{c,1}, \ldots, r_{c,t}\}$. To perform the maintainability assessment, we assume three types of maintenance activities, i.e. adding new components, adding new plug-ins to existing components and changing existing component code. Each of these activities has an associated productivity measure, i.e. P_{nc}, P_p and P_{cc}, respectively. Based on earlier research, e.g. [9] and [13], and our own experience the productivity when developing new components for an existing system is at least an order of magnitude higher than changing the code of existing components.

Once the maintenance profile is available, we can perform impact analysis, i.e. analyse the changes to the software architecture and its components required to incorporate the change scenarios. The result of the impact analysis process is a set of impact analyses, one for each change scenario, i.e. $IA = \{ia_1, \ldots, ia_u\}$, where ia_1 is defined as $ia_i = \{CC_i, NP_i, NC_i, R_i\}$. Here, CC_i is the, possibly empty, set of components that need to be changed for incorporating the change scenario, including a size estimate for the required change in lines of code, $CC_i = \{\{c_j, s_j\}, \ldots\}$. NP_i and NC_i contain similar information for new plug-ins and new components, respectively. R_i contains the new, changed and removed relations required for incorporating the change scenario.

As a final step, once the impact analysis has been performed, it is possible to determine the maintenance effort required for the maintenance profile. This can be calculated by summing up the efforts required for each change scenario. Below, the equation is presented:

$$Maintenance\ effort$$
$$= \sum_{IA}\left(\left(\sum_{CC_i} s_j\right) \cdot P_{cc} + \left(\sum_{NP_i} s_j\right) \cdot P_p + \left(\sum_{NC_i} s_j\right) \cdot P_{nc}\right)$$

(Equation 1)

4. Assessing Optimal Maintainability

Once a quantitative assessment of maintainability has been performed, the next question a software architect typically will ask is: “Does this mean that the maintainability of my architecture is good, reasonable or bad?”. Therefore, it is important to be able to compare this to the optimal maintainability for the scenario profile and the domain. We have developed a technique that allows us to obtain this information.

To assess optimal maintainability, we make use of the maintenance profile, *MP*, and the results of the impact analysis, *IA*. The technique is based on two assumptions. First, the amount of code that needs to be developed or cahnged for a change scenario is relatively constant in size, independent of the software architecture. Second, the main difference in maintenance effort is due to the difference in productivity between different maintenance activities, i.e. $P_{nc} < P_p \ll P_{cc}$. For example, productivity being (>6x) higher when writing new code (>250 LOC/Man-month in C [13]) compared to modifying legacy code (1,7 LOC/ Man-day ~ <40 LOC/Man-month [9]). Note that these productivity metrics include more than simply writing the source statements.

Based on these assumptions, it is clear that an optimal software architecture would allow for the incorporation of all change scenarios by adding new components or, in the case of smaller change scenarios, by adding new plug-ins for existing components.

Thus, a straightforward approach is to calculate the required effort assuming that all necessary changes can be implemented as part of a new component. Below, the equation calculating this is presented:

$$Optimal\ maintenance\ effort$$
$$= \sum_{IA}\left(\left(\sum_{CC_i} s_j\right) + \left(\sum_{NP_i} s_j\right) + \left(\sum_{NC_i} s_j\right)\right) \cdot P_{nc}$$

(Equation 2)

Although this indeed calculates a lower boundary for the required maintenance effort, two issues remain unresolved. First, especially for change scenarios requiring a relatively small change in terms of code size, it may not be reasonable to assume that these are mapped to components. Such change scenarios should be mapped to new plug-ins and the associated productivity metrics should be used. The second issue is a more subtle one. The straightforward model assumes that it is possible that for a maintenance profile, all change scenarios can be mapped to either a new component or a new plug-in. The question is, however, whether it is possible to define a software architecture that supports this. Especially in the case where different change scenarios address the same requirements, it is unreasonable to assume that these change scenarios

$$Opi\ maint.\ effort = \sum_{IA} \left(\left(\left(\sum_{CC_i} s_j \right) + \left(\sum_{NP_i} s_j \right) + \left(\sum_{NC_i} s_j \right) \right) \cdot \begin{cases} P_{nc} & \textit{if independent \& large scenario} \\ P_p & \textit{if indepdendent \& small scenario} \\ (P_{cc} \cdot ratio + P_{nc} \cdot (1 - ratio)) & \textit{if interacting \& large scenario} \\ (P_{cc} \cdot ratio + P_p \cdot (1 - ratio)) & \textit{if interacting \& small scenario} \end{cases} \right)$$

(Equation 3)

can be implemented independently, i.e. we experience interaction of change scenarios.

Based on the discussion above, we refine our technique for the assessment of optimal maintainability as follows. First, we categorize change scenarios as independent or interacting. A change scenario is independent if it affects existing requirements in the requirement specification that are not affected by another change scenario that has a lower index number in the maintenance profile than the current scenario. This results in the situation that one change scenario can affect an existing requirement by factoring it out as a separate component or plug-in, but all subsequent change scenarios affecting the same requirement are considered to be interacting.

An interacting change scenario affects one or more requirements in the requirement specification that are also affected by change scenarios with a lower index number. The effort required for this type of change scenarios is calculated based on the ratio of interacting and independent requirements in the set affected by the change scenario. Thus, if, for instance, two of five existing requirements for a change scenario are interacting, then 40% of the total change size (in, e.g., lines of code) is calculated using the P_{cc} productivity metric. The remainder is calculated by either using the P_{nc} or the P_p productivity metric.

If the total change size of the independent part of the change scenario is less than the code size of the smallest architectural component and/or less than half of the average code size of architectural components, then the P_p productivity metric will be used to calculate the effort. Otherwise, the P_{nc} productivity metric is used.

Thus, for each type of change scenario, the required effort can now be calculated. In equation 3, the alternatives are presented. The term 'ratio' refers to the ratio between interacting and independent scenarios. The total maintenance effort can be calculated by summing up the required effort for each change scenario.

5. Illustrating the Technique

The system that we will use to illustrate this technique is a cellular telecom fraud detection and management system called FCC that has been developed by Ericsson Software Technology AB, Sweden. Usually, cellular telecom operators introduce cellular telephony into an area with a primary concern with establishing network capacity, geographic coverage and signing up customers. However, as subscriber and network growth level cost margins and other financial issues become more important, e.g. lost revenues due to fraud.

Fraud in cellular telecom can be divided into two main categories; subscriber fraud and cloning fraud. Subscriber fraud is when subscribers use the services but does not pay their bills. The two main means to prevent this is to perform subscriber background and credit history checks. Cloning fraud means that a caller uses a false or stolen subscriber identification in order to make calls without paying, or to be anonymous. There exist various approaches to identify and stop cloning. The FCC system is a part of an anti-fraud system that counter-checks cloning fraud using real-time analysis of network traffic.

The FCC system is a commercial system and because of this we cannot disclose the actual requirements posed on it. For the purpose of illustrating the method we will denote the requirements as R1, R2, etc. We are aware that this is a limitation of the illustration, but argue that the problem is inherent to using a contemporary commercial system. The alternative would be to use a artificial example, or a retired system, but it too has its limitations. In addition, because of size restrictions we are not able to present all details in this illustration, e.g. the complete set of scenarios, or an elaborate description of the impact analysis itself.

5.1.System Overview

Software in the switching network centres provides real-time surveillance of suspicious activities associated with a call. The idea is to identify potential fraud calls and have them terminated. However, one single indication is not enough for call termination.

The FCC system allows the cellular operator to specify certain criteria for terminating a suspicious call. The criteria that can be defined are the number of indications that has to be detected within a certain period of time before any action is taken. It is possible to define special handling of certain subscribers and indication types. For the rest of this paper we will use the term 'event' to denote a fraud indication from the switching network.

The events are continuously stored in files in the cellular network. With certain time intervals or when the files contain a certain number of events these files are sent to the FCC. The FCC system stores the events and matches them against pre-defined rules. When an event match a rule, the FCC system sends a message to terminate to the switching network and the call is terminated. It is possible to define alternative actions, e.g. to send a notification to an operator console.

Table 1. The Assessment Results of the Architecture

SCENARIO	COMPONENT												NEW COMPONENTS				AFFECTED REQUIREMENTS
	A		B		C		D		E		F		G	H	I	J	
	1 kLOC		4 kLOC		1 kLOC		3 kLOC		5 kLOC		1 kLOC		2 kLOC	5 kLOC	2 kLOC	.5 kLOC	
1			.2	.5			.1	.1	.2								r2, r10
2	.3		.1				.2										r3
3	.5	.5			.8	.5											
4							.2		.3				1				r4
5							.4		.5								
6	.5	.5					.3	.3						1			r1,r11,r12
7							.1	.1			.1	.8					r5,r13,r14,r15
8	.2	.2					.4	.4			.1	.5		1			r1, r5, r16
9			.3	.4			.5	.2	.3	.3							r2, r17, r18
10	.3														1		r2
11											.2	.2					r5
12	.2															1	r8
13							.3	.5	.2	.3							r9

5.2. Architecture

The FCC application consists of seven software components, all executing on the same computer (see Figure 2). The majority of the components was designed using object oriented techniques and implemented in C++. In one component a scripting language was used in addition to C++. The components do not follow a predifined component model but are rather collections of related classes, a view of components that is common in industry [7]. *TMOS* is an Ericsson propriety off the shelf component that handles the interaction with the switching network. In this application it is used for collecting event files from switches and other utility services. The *Listener (a)* collect events from switches via the TMOS component. The TMOS component is a third party component and is considered only as is, i.e. we cannot make changes to this component. The events are then propagated to one of the executing *Parser (b)* threads. The parser extracts the event data from the event files and store then in the *EventQueue (c)*. Using predefined rules the *DBClient (d)* checks the events in the Event Queue and stores the event data in the database. If an event triggers a rule, the *Action (f)* component notified, who in turn executes the actions associated with the rule, e.g. a call termination. Standard Unix scripts are used to send terminating messages to the switching network. The components marked *g-j* are components that were judged as potential new components during the assessment, and have been added to this figure to illustrate how the fit in the rest of the architecture.

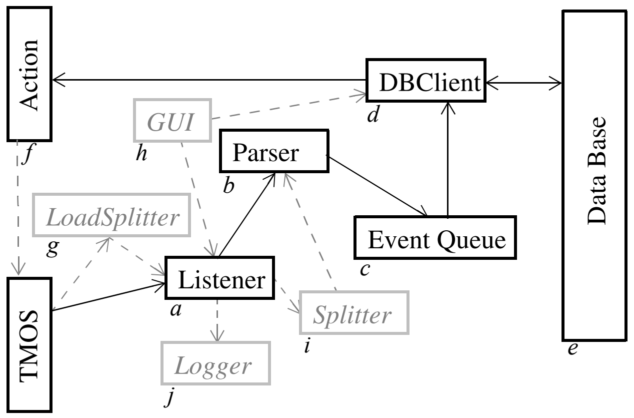

Figure 2. The Main software architecture

5.3. The Scenario List

For the purpose of this study we asked a person related to the FCC project to prepare a list of the most likely changes to the FCC for the future. No other restrictions or instructions were given. We present a selection of the scenarios from the list we received and used in the assessment below.

S1 New version of the switch software. Could lead to new event types and changed attributes of events.

S2 Port to Windows NT.

S5 Change of data base supplier.

S6 Port of graphical user interfaces to Java.

S8 Interactive termination. An operator must acknowledge call termination.

5.4.The Results of Scenario-Based Assessment

The original architecture (figure 2) was assessed with the assessment method described in section 3. The components are marked in the figure 2. The results from the scenario impact analyses are presented in the table below (Table 1). For this illustration we choose to set productivity for changing (P_{cc}) to 40 LOC/Man-month [9], and based on our assumption that writing plug-in code is somewhat easier to do than modifying existing code, we choose to set P_p at 60 LOC/Man-Month. Based on the productivity for writing new code we choose to set P_{nc} at 250 LOC/Man-Month [13].

The maintainability effort on average per scenario of this architecture according to the calculation in Equation 1 and based on the changes scenarios defined in section 5.3 is 63 man-months. The optimal maintenance effort according to the first equation (Equation 2) is 16 Man-Months on average per scenario. The refined equation renders these results instead (Equation 5) is 27 Man-Months on average per scenario.

Concluding, the technique we present in this paper allows the software architect to compare the assessed value (63 man-months per change scenario) with the optimal value (27 man-months per change scenario). Based on the difference between these figures, the architect may decide whether this is acceptable or that additional effort has to be spent on improving on the software architecture for maintainability. Since software architecture design requires trade-offs between different quality attributes, the presented technique provides the software architect with more and better information in order to take well-founded decisions.

6. Related Work

The software architecture analysis method (SAAM) [11] was among the first to address the assessment of software architectures. SAAM employs stakeholder-based assessment to determine the suitability of the software architecture for the system at hand. The stakeholders define scenarios that are used to evaluated the software architecture.

An example of quality attribute oriented analysis technique is the architecture trade-off analysis method (ATAM) [12]. ATAM is the successor of SAAM. The method defines attribute models that are used to identify trade-off points in the architecture. These points are potentially problematic areas in the architecture that, for instance through redesign, can be neutralized.

In [8], the authors present a software architecture evaluation model that performs quality attribute evaluation based on the ISO/IEC 9126 draft standard. The approach employs metric models to determine internal and external metrics of the system that can be used to evaluate the quality requirements.

Some specific quality attribute assessment techniques have been developed. [1] discuss an approach to assess the timing properties of software architectures using a global rate-monotonic analysis (RMA) model that is composed from component RMA models. This allows the architect to early obtain information about the real-time behaviour of the architecture. In [3], we propose a technique for the assessment of the maintainability of software architectures. This technique was discussed in section 3.2. This assessment technique is part of a software architecture design method [6].

None of the discussed work proposes techniques for the assessment of optimal quality attribute levels, which is the topic and contribution of this paper.

7. Conclusions

The work presented in this paper is motivated by the increasing realization in the software engineering community of the importance of software architecture for fulfilling quality requirements. Traditionally, the software architecture design process is rather implicit and experience driven without a clear and explicit method. One main reason for this is the lack of architecture assessment techniques that quantify quality attributes of the software architecture.

In earlier work [3], we presented a scenario-based technique for assessing the maintainability of a software architecture. The technique employs a scenario profile, used for performing impact analysis and calculates maintainability based on this. Although very useful, the software architect is typically interested in the optimal maintainability level in order to understand how close the actual level is to the optimal level. In this paper, we presented a technique for assessing the optimal maintainability for the software system, based on the scenario profile and impact analysis. The technique determines the required maintenance effort assuming that all new and changed requirements can be implemented by adding new components or plug-ins to the system.

We have illustrated and exemplified the technique by using an industrial case from the telecom domain and found that, according to the method proposed in this paper, the optimal average effort per scenario is about half of the same average effort per scenario of the current architecture. In addition, the most optimistic calculation (Equation 2) suggested only a quarter of the average effort compared to the current architecture. This, however, seem unrealistically optimistic.

In the future we intend to continue with the validation and refinement of the technique by applying it to additional industrial cases and investigating the validity of the assumptions made. In addition, we are interested in investigating whether the technique can be applied to other quality attributes as well.

Acknowledgements

We thank Ericsson Software Technology for providing us with the FCC case and allowing us to publish these results.

References

[1] A. Alonso, M. Garcia-Valls, J.A. de la Puente, “Assessment of Timing Properties of Family Products,” Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, F. vd. Linden (editor), LNCS 1429, pp. 161-169, 1998.

[2] L. Bass, P. Clements, R. Kazman, *‘Software Architecture In Practise’*, Addison Wesley, 1998.

[3] P. Bentsson, J. Bosch, ‘Architecture Level Prediction of Software Maintenance’, *The 3rd European Conference on Software Maintenance and Reengineering* (CSMR'99), pp. 139-147, 1999.

[4] P. Bengtsson, J. Bosch, ‘Haemo Dialsis Software Architecture Design Experiences’, in Proceedings of International Conference on Software Engineering (ICSE’99), ACM Press, New York, pp. 516-525, 1999.

[5] P. Bengtsson and Jan Bosch, “An Experiment on Creating Scenario Profiles for Software Change”, *Annals of Software Engineering*, Vol. 9, May 2000.

[6] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, May 2000.

[7] J. Bosch, ‘Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. Proceedings of the First Working IFIP Conference on Software Architecture, February 1999.

[8] J.C. Duenas, W.L. de Oliveira, J.A. de la Puente, “A Software Architecture Evaluation Model,” Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, F. vd. Linden (editor), LNCS 1429, pp. 148-157, 1998.

[9] J. E. Henry, J.P. Cain, *"A Quantitative Comparison of Perfective and Corrective Software Maintenance"*, Journal of Software Maintenance: Research and Practice, John Wiley & Sons, Vol 9, pp. 281-297, 1997

[10] Daniel Häggander and Per-Olof Bengtsson, Jan Bosch and Lars Lundberg, Maintainability Myth Causes Performance Problems in Parallel Applications, Proceedings of *IASTED 3rd International Conference on Software Engineering and Applications,* pp. 288-294, October 1999.

[11] R. Kazman, L. Bass, G. Abowd, M. Webb, ‘SAAM: A Method for Analyzing the Properties of Software Architectures,’ *Proceedings of the 16th International Conference on Software Engineering*, pp. 81-90, 1994.

[12] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, ‘The Architecture Tradeoff Analysis Method,’ *Proceedings of ICECCS’98,* (Monterey, CA), August 1998.

[13] K. D. Maxwell, L. Van Wassenhove, S. Dutta, *"Software Development Productivity of European Space, Military, and Industrial Applications"*, IEEE Transactions on Software Engineering, Vol. 22, No. 10: OCTOBER 1996, pp. 706-718

[14] D.E. Perry and A.L. Wolf, “Foundations for the Study of Software Architecture”, ACM SIGSOFT Software Engineering Notes, pp. 40-52, 1992.