International Journal of Software Engineering and Knowledge Engineering
Vol. 10, No. 1 (2000) 97–114
© World Scientific Publishing Company

AN APPROACH TO QUANTITATIVE SOFTWARE ARCHITECTURE SENSITIVITY ANALYSIS

CHUNG-HORNG LUNG and KALAI KALAICHELVAN
Software Engineering Analysis Lab (SEAL), Nortel Networks,
P. O. Box 3711, Station C, Ottawa, Ontario, Canada K1Y 4H7
e-mail: lung;kalai@nortelnetworks.com

Software architectures are often claimed to be robust. However, there is no explicit and concrete definition of software architecture robustness. This paper gives a definition of software architecture robustness and presents a set of architecture metrics that were applied to real-time telecommunications software for the evaluation of robustness. The purpose of this study is to provide a structured method to support software architecture evaluations and downstream software implementations. The study also expands the software architecture research to quantitative and measurable evaluations as opposed to qualitative assessments. In addition, this paper presents an empirical case study of applying the metrics. The approach and the metrics data provide insights into software architecture sensitivity analysis on system qualities and trade-off analysis among a set of design alternatives to support product evolution.

1. Introduction

Software architecture has drawn tremendous attention in the past several years in both academia and industry. The main reasons are the increasing complexity of software systems, the need to analyze and design systems at higher levels of abstraction and the demand to reduce maintenance costs for system evolution. However, software architecture properties are usually claimed through abstract terms such as "The architecture is robust". Further, software architecture analyses are often conducted in an *ad-hoc* manner. This paper presents an approach to facilitate structured evaluation of software architectures for product evolution and trade-off analysis.

To evaluate an architecture in an objective and repeatable manner, and to establish a basis for effective communication within the software architecture community, a definition of robustness is necessary. Kalaichelvan and Munikoti presented the concept of software architecture robustness [10]. This paper follows their school of thought and extends the work by identifying more concrete factors that impact software architecture robustness.

Moreover, explicit evaluation is needed through measurable metrics. There are numerous metrics available at the code level [14]. Some researchers have also worked on the design metrics [1]. However, there is very little emphasis, if any, on the metrics of higher level of abstraction at the architectural stage. This paper is one of the first to attempt to design various metrics for quantitative evaluation of software architectures from the evolution perspective. It is demonstrated, in this paper, that quantitative evaluation can be performed to support decision making process for the software practitioner.

This paper is organized as follows. Section 2 presents a definition of software architecture robustness. Section 3 describes a list of metrics at the architecture level and presents an example. Section 4 discusses an approach to collect the metrics data and explains how to quantitatively measure software architecture robustness with two case studies. Section 5 shows some related research in software architecture trade-off analysis. Section 6 presents some lessons learned from empirical studies. Finally, Sec. 7 shares the current status of this research and identifies future directions in this area.

2. Software Architecture Robustness: A Definition

Software architecture robustness has been constantly addressed and recognized as a critical issue. But what is software architecture robustness? There is no explicit or concrete definition of robust software architecture. Here we propose a definition of software architecture robustness and a set of metrics for making quantitative as well as qualitative evaluations.

The definition is based on analyzing the sensitivity of a software architecture to the changes in critical customer objectives. Specifically, software architecture robustness is defined as the degree of sensitivity of the architecture to the effects of change in stakeholder value parameters measured in terms of architecture attributes. Figure 1 demonstrates the concepts and some example factors that affect the robustness of a software architecture. Based on the definition, the robustness is related to the objective of the stakeholder, customer value parameters, and architectural concerns and attributes. Typical stakeholders are customers, architectural evolution strategists and chief architects of the organizations. Examples of customer value parameters include quality attributes like scalability, performance, and modifiability, and architectural concerns such as conformance to standards and consistent interfaces. Key architecture attributes are the critical software architectural elements, including components and connections.

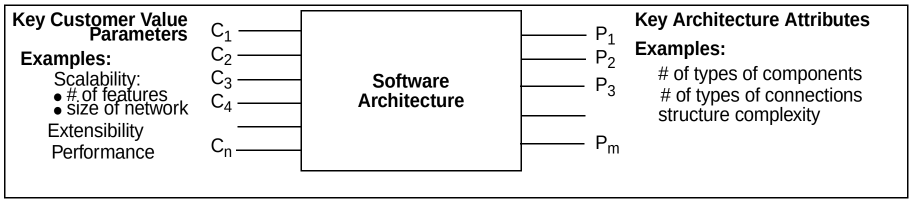

Fig. 1. Software architecture robustness analysis factors.

In other words, if an architecture is designed to accommodate many changes, then the architecture is robust. If some architectural elements that must be changed do not have much impact on the architecture or the downstream implementation, then the architecture is also robust, because the architecture is independent of, or not sensitive to the changes. To support a quantitative analysis of sensitivity or robustness, a set of metrics are needed to describe the magnitude or degree of changes. The following section depicts a set of metrics for robustness or sensitivity analysis.

3. Software Architecture Metrics

Perry and Wolf [21] presented a software architecture model, which consists of three basic classes of architecture elements. These classes are processing element, data element, and connecting element. Before presenting the metrics, we assume that the software architecture is represented in such a way that components and links are classified and/or generalized [7,11]. For example, the components could be classified into computational units, process units, threads, and data repositories. Links between components could be classified into control links and data links. Moreover, a number of components and links could be logically grouped to get a higher level of abstraction or to form an architectural style [7]. Figure 2 presents a simplified call processing software architecture which illustrates the concept of classification and generalization of components and links. Processes represent independent threads of control. A process may consist of a set of tasks that form an executable unit. Computational units only exist within a a process or within another computational unit. An active data repository consists of both data and methods that manipulate the data. Passive data repositories represent only data, e.g., traditional files. A persistent component statically stays in the system, whereas non-persistent components are instantiated dynamically at run-time. The description here only provides a guideline, as the classification of the components and connections is application-specific. There are many metrics developed for the code level. Generally speaking, the code metrics can be categorized into measurements of

- Size (lines of code, number of functions, number of classes)
- Structure (cyclomatic number, fan-in and fan-out)
- Usage (local variable vs. global variable) [6]
- Levels of abstraction (procedure or class level vs. file level)

Although many of the code metrics, such as lines of code, will not be applicable at the architecture level directly, the categories of measurements are useful for higher level as well, such as architecture. A set of metrics for software architecture is developed based on these categories. Counting of components is related to “size”. Computational units and process units are, in general, more complicated than data repositories. This feature indicates the complexity of the internal “structure” of a component, which shares similarity with the cyclomatic metric at the code level. In addition, the overall topological "structure" of an architecture bears the information of fan-in and fan-out of components and subsystems. Different components and links may have different "usages". For instance, components could be persistent or non-persistent, and links could be control or data. For "levels of abstraction", the components could be logically grouped to demonstrate the system at a higher-level of detail, i.e., logical grouping or architectural style. The concept is used for both procedural languages (procedure level vs. file level or file level vs. module level) and object-oriented languages (class level vs. file level).

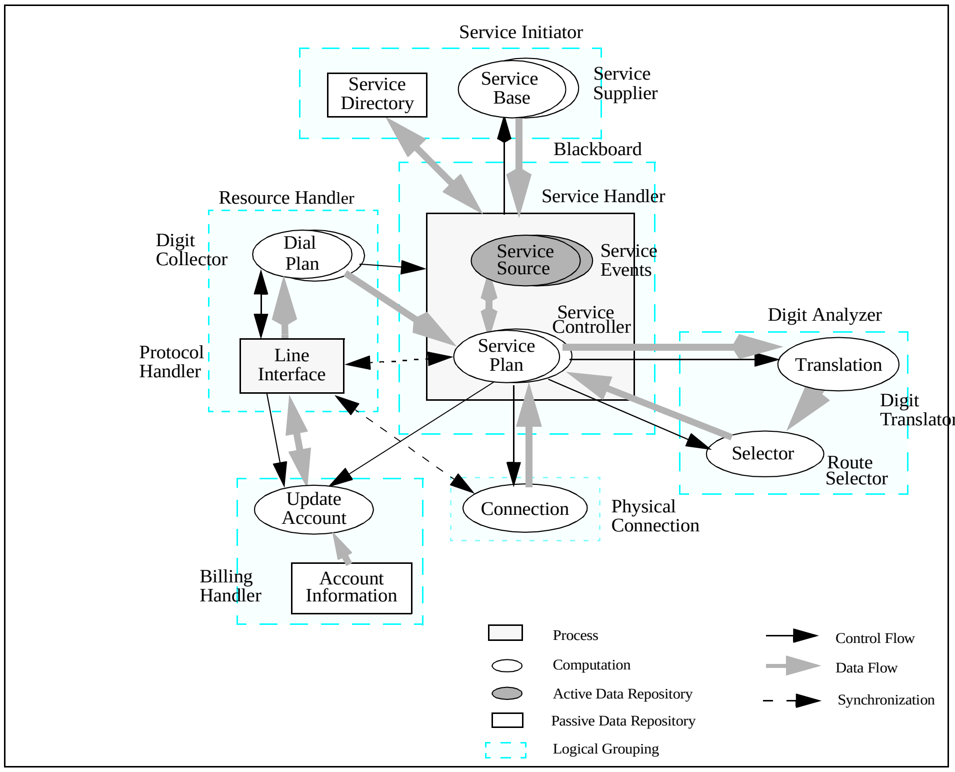

Fig. 2. Structural view for a simplified call processing system.

Table 1 presents a set of metrics for software architecture analysis. The entities and values shown in the table are extracted from the example illustrated in Fig. 2. The metrics of CL (coupling link) type deals with the coupling aspect for each component. There is a value associated with each individual component of type CL, as shown in the Entity column. The ratio of function/component shows the mapping between functions and components. A function which involves many components may be a sign that the particular function is complicated, the set of components are highly coupled, or some of the components are at a fine-grained level. Higher

Table 1. Metrics for measuring software architecture.

Name	Type	Metric	Entity	Value
NCU	C	# of computational units	Dial Plan, Service Base, Service Plan, Translation, Selector, Connection, Update Account	7
NPU	C	# of process units	Line Interface, Service Handler	2
NADR	C	# of active data repositories	Service Source	1
NPDR	C	# of passive data repositories	Service Directory, Account Information	2
NPC	C	# of persistent components	Line Interface, Service Base	2
NNPC	C	# of non-persistent components	the rest, excluding regular files	9
RPC	C	ratio of persistent components/total # of units		2/11
RNPC	C	ratio of non-persistent components/total # of units		9/11
RCP	C	ratio of (computational + process)/total # of units		9/11
NCL	DCL	# of control links		9
NDL	DCL	# of data links		11
NSL	DCL	# of synchronization links		2
NASL	DCL	# of asynchronization links		0
NTCM	DCL	# of type of communication mechanisms	memory read/write, CORBA, TCP/IP(not shown in Fig. 2)	3
NFOP	CL	# of fan-out of process units	Line Interface (6), Service Handler (10)	16[b]
NFIP	CL	# of fan-in of process units	Line Interface (4), Service Handler (8)	12[b]
NFOC	CL	# of fan-out of computational units	Service Base (2), Dial Plan (3), Service Plan (7), Translation (1), Selector (1), Connection (2), Update Account (1)	17[b]
NFIC	CL	# of fan-in of computational units	Service Base (2), Dial Plan (2), Service Plan (8), Translation (2), Selector (2), Connection (2), Update Account (4)	23[b]
MAXRFC	F	max. ratio of function/components	Function Digit Collection (Dial Plan and Line Interface)	1/2
MINRFC	F	min. ratio of function/components	Function Physical Connection (Line Interface, Service Source, Service Plan, Selector, Connection)	1/5
ANFC	F	average # of function/components		1/3
NLG	AL	# of logical grouping (cluster)	Service Initiator, Service Handler, Resource Handler, Digit Analyzer, Physical Connection, Billing Handler	6
NAS	AL	# of architectural styles or patterns	blackboard	1
NVAS	AL	# of violations to architectural styles	Line Interface to Connection	1
SC	AL	Structure complexity [5]		16.8
SCC	AL	System cohesion & coupling [8]		0.56

[a]C: Component; DCL: Data or Control Link; CL: Coupling Link; F: Function; AL: Architecture Level
[b]A value for each individual component is identified.

coupling may be necessary for some components due to their specific functionalities, such as shared utilities. For these type of components, decoupling may not be necessary. However, the point is to capture those features explicitly for better understanding.

Note that these values are raw metrics. The values listed in Table 1 at this stage do not indicate the robustness of an architecture nor are intended to be used for comparing different architectures directly. Moreover, these metrics are highly multi-colinear. In other words, trying to change one value of a metric through architectural modifications will affect other metrics. Instead, we are more concerned with the magnitude of changes or delta values of the metrics for the sensitivity analysis rather than the raw values. The delta analysis is used for evolution planning or comparing different software architectures in the same problem domain. The following section describes how to collect and make use of these metrics to measure the sensitivity, and hence robustness.

4. Software Architecture Robustness Analysis

The metrics shown in the previous section presents a "static" view of a software architecture. These metrics were developed based on studies of large-scale real time telecommunications systems. However, the metrics are applicable to other application domains. A software architecture robustness analysis was performed using these metrics. In order to measure the sensitivity or robustness of a software architecture with the metrics, we adopt and expand a scenario based approach [11,12,18] to collect and process the data. The approach extends the Goal/Question/Metric paradigm [2] by considering a wider range of stakeholders and their objectives. A number of use cases or explicit scenarios are derived from the key stakeholder objectives or customer value parameters. A use case is a category of transactions performed by a system, describing a particular purpose of using the system or yielding an observable result from the actor's point of view [9]. A scenario is a use case instance. In addition, a set of architectural views were adopted to support the analysis [15–18]. By walking through the architecture with explicit scenarios derived from the key stakeholder objectives or customer value parameters, we can identify the changes required due to new, modified, or missing requirements. The information is then mapped to the set of metrics to get qualitative or even quantitative results. To highlight, the approach iteratively applies the Objective/Scenario/Metric paradigm, which is described as below.

- **Objective.** This phase gathers and analyzes different types of information, namely, stakeholder objectives, architectural objectives, and non-functional quality objectives such as maintainability, reliability and performance [3,18].

- **Scenario.** A set of scenarios or use cases are derived from the stakeholder objectives, architectural objectives, and desired system quality attributes. The scenarios cover both the expected usages of a system, predicted management, system behaviors for abnormal conditions, and potential future changes to a system. A use case or scenario classes could be further refined into more concrete or more detailed scenarios.
- **Metric.** A set of metric values is identified based on the scenario analysis to provide quantitative answers. The metrics are primarily used to evaluate the effort required for system evolution or the sensitivity of an architecture based on the analysis.

4.1. *Case Study 1*

Table 2 illustrates the concept and presents an example on the kinds of information that is gathered for a large telephone switching system. Three objectives are listed to depict some anticipated future usages. For each objective, a set of scenarios are developed. Table 3 highlights some example scenarios developed based on the stakeholder and architectural objectives. Elicitation questions are prepared for each objective and are used in interviewing domain subject experts. These interviews are used to better understand systems and to develop scenarios for analysis.

Table 2. Stakeholder – Architectural – Quality objectives: An example.

Stakeholder objectives	Architectural objectives	Quality impact
Support terminal, user, and session mobility	Decouple and encapsulate all terminal, user, and session data, and permit the transport of data. Allow terminal, user, and session data to be configured dynamically.	Reliability Interoperability Portability Scalability
Support multimedia simultaneously within the same session.	Provide consistent common interfaces to all media. Provide scalable communication media. Separate product functionality from hardware constraints.	Reliability Modifiability Interoperability Performance
Provide flexibility to port to a multi-processor environment.	Decouple functionalities. Use common interfaces among components. Encapsulate the transport of data and make no assumption of explicit physical information.	Reliability Modifiability Interoperability Performance

Each objective may consist of a set of scenarios. Moreover, the scenarios developed for each objective could be categorized for complex applications, creating a reusable checklist of architectural concerns. In telecommunications systems, for instance, interactions of complex services or features need to be validated. Those feature interactions are grouped into different classes to have better scenario coverage and to facilitate evaluations. Another important example is fault management. There are many possible types of faults. The category is useful for scenario generation, validation, and coverage analysis [4,18]. By working with the stakeholders closely, we can also prioritize scenarios, especially when there is a timing or resource constraint.

Table 3. Scenarios: An illustration.

Stakeholder objectives	Scenarios
Support terminal, user, and session mobility	• Terminal mobility. A person needs to access a non-fixed link to the network. Examples include cellular phones or mobile computer connections to the network. • User mobility. A user can use another user's terminal device as his/her own. This requires that the user is identified and functionalities, including billing policy, can support it. • Session mobility. A user may move a particular call from one terminal device to another. For instance, a person using a celluar phone wants to transfer the call to another terminal device and continue the call.
Support multimedia simultaneously within the same session.	• Two persons can establish a call to talk, send a fax, and discuss some thing displayed on a computer screen simultaneously within the same session.
Provide flexibility to port to a multi-processor environment.	• The called party and the calling party are to be handled by separate processors.

In addition to the scenarios developed directly from objectives, a group of scenarios for basic uses of the system may need to be generated. Often, analyses will focus on potential future changes to a system. Basic needs are thus usually neglected. Basic needs usually are not product differentiators, yet one cannot have a product without the basic functionality. For example, a basic call service must exist no matter how complex the communications may be. Basic needs are thus critical for architectural analysis, but often are not explicitly expressed by stakeholders.

For each scenario, the effect on the architecture is identified. Typically, there is either no effect (no change to the architecture required) since the scenario is directly supported by the architecture, or changes in the architecture are required to satisfy the scenario. In addition, the effort required to make the necessary changes is also estimated based on the types of changes and components. Issues for further analysis are addressed if more specific information is needed to perform the analysis.

Table 4 demonstrates an example of partial analysis results. The existing system serves as a reference architecture. Having a reference architecture is critical to make more concrete or meaningful comparisons among alternatives. The numbers shown in the table indicate the effects for the potential changes with respect to the reference architecture in order to meet the objectives. The approach coupled with the metrics support architecture-driven evolution, especially when the table gets populated. The structured method of identifying information and gathering metric data provides insights into change impact analysis and system evolution. For each objective, the changes that need to be made are captured and classified. Combination of objectives can reveal frequency of change for components or links. The recurrence of some architectural elements in the table is a sign of change prone areas, which usually require more maintenance than other areas.

The information may also provide positive side effects. The results could indicate weaknesses of a product or the process. For instance, many links need to be modified may mean that the communication mechanism needs to be simplified or re-examined. If many changes point to the same source (e.g., requirements gathering or design), this information may support the improvement of process.

Table 4. Changes[1] required to support scenarios derived from business objectives: An illustration.

Metric	Objective #1 support mobility	Objective #2 support multimedia	Objective #3 support multiprocessing
# of computational units	+2: Medium (Service Plan, Connection) +1: Low (Update Account)	+1: High (Connection) +1 Low: (Service Base)	+2: High (Service Plan, Connection) +2 Medium (Translation, Selector)
# of process units	+1: High (Line Interface) +1 Low: (Service Handler)	+1: High (Service Handler)	+2: High (Line Interface, Service Handler)
# of active data repositories	0	+1: High (Service Source)	0
# of control links	+1: Medium (Line Interface to Dial Plan) +1: Medium (Line Interface to Service Plan)	+1: Medium (Selector to Connection)	+2: High (Line Interface to Service Handler (2) +2: Medium (Service Plan to Connection, Service Plan to Translation)
# of types of communication mechanisms	−1 (CORBA)	−1 (CORBA)	−1 (CORBA)
# of violations to architectural style	+1: Low (Selector to Connection)	+1: Medium (Dial Plan to Translation)	0
Structure complexity [5]	0 (same)	−1 (from 16.8 to 18.5)	−2 (from 16.8 to 22.3)
System strength [8]	0 (same)	0 (from 0.58 to 0.55)	−1 (from 0.58 to 0.65)

[1]A positive number indicates the number of changes (modifications or additions) needed. A negative number indicates the number of removal of components or links. 0 means no evident changes required. High, Medium, and Low represent the level of estimated effort necessary to make the particular change.

Other information may be derived from the matrix. For instance, we could easily obtain the ratio of components that need to be modified to the total number of components for each stakeholder objective. The information could also be divided into categories according to the level of estimated effort required to make the changes. If the matrix (Table 4) is highly sparse, then the architecture is not sensitive, and hence robust according to the definition of robustness, for a given set of objectives. If the matrix is highly populated or the number of changes is large, then the architecture is not robust with respect to the objectives. The analysis should be performed by walking through the architecture with a wide range of scenarios, which in turn are developed to evaluate against the stakeholder objectives. Modeling of stakeholder objectives and generating scenarios can be found in [4,18].

The analysis results described in Table 4 show the number of modifications that need to be made and the estimated effort necessary to make the changes. Further, some changes may be independent or dependent on other changes. The approach also indicates volatile areas (components, links, or mechanisms) where changes are likely to occur. The approach and metrics together with other application-specific information support the identification of change interdependencies, which could provide insight into prioritizing and scheduling activities in a release cycle.

Another often encountered problem is possible conflicts that will be introduced when making changes. For instance, we may add a link between two components to improve performance. But this modification often has negative impact on maintainability. Table 4 shows some potential risks from the stakeholder objectives point of view. The analysis we propose also extends to evaluate the system based on non-functional attributes by identifying the impact on qualities for each change or change class. Table 5 demonstrates the concept with an example. The first three columns represent three instances of change for either a component or a link, whereas the last column shows a change for one type to another. Impact values are in the range of -2 to $+2$ for the proposed changes. The values are obtained by a team of domain experts. As shown in Table 5, the proposed change 'add services to existing *Line Interface* to support mobility' has a great deal of negative impact on qualities. As a result, further investigation should be conducted to explore other means to support mobility or restructure exising software architecture. The impact analysis based on the structured process has been empirically validated as an effective vehicle for communications between stakeholders, which is critical in product evolution planing.

The approach is further used to support trade-off analyses among design alternatives or support concept selection for better decision-making. For instance, if there are several candidates for a proposed change, we could evaluate these candidates by applying the approach and compare the candidates against a set of selection criteria that are critical to the stakeholders. This method is adapted from Ulrich and Eppinger [22]. An example is depicted in Table 6 for illustration.

The example takes the previous identified risk area described in the last paragraph one step further by examining other possible ways to modify the *Line Inter-*

face to support mobility. The first method, 'add services to existing *Line Interface*', is taken from the previous step for comparison purpose. Another method is 'decouple *Line Interface* from protocol handling' and develop a separate component for this feature. The third one is an extension of the second one by introducing an additional component to handle user profiles. Based on the reference architecture, the team then rates these three alternatives against the selection criteria. Note that the criterion Time to Market is evaluated relatively rather than based on the common reference architecture. If the system starts from scratch, one of the design options can be chosen as the reference architecture.

Table 5. Trade-off analysis for quality impact: An illustration.

Quality impact	Add services to existing Line Interface to support mobility	Divide Service Handler into two for both parties	Add a control link from Selector to Connection	Replace CORBA with Memory Read/Write
Reliability	0	+1	0	+1
Maintainability	−1	+1	−1	+1
Interoperability	−1	+2	−1	−1
Portability	−2	+1	−1	−1
Scalability	−1	+1	−1	−1
Performance	0	−1	+2	+1

Table 6. Evaluation of architectural alternatives.

Selection Criteria	Add services to existing Line Interface	Decouple Line Interface	Decouple Line Interface plus add user profile
Reliability	0	+1	+1
Maintainability	−1	+1	+1
Interoperability	−1	+1	+2
Portability	−1	+1	+2
Scalability	−1	+1	+1
Performance	0	−1	−1
Time to Market	+1	0	−1
Sum +'s	1	5	7
Sum 0's	2	1	0
Sum −'s	4	1	2
Net Score	−3	4	5
Rank	3	2	1
Continue?	No	Yes	Yes

The last row in Table 6 shows a net score for each alternative by subtracting '−' ratings from the '+' ratings. At this stage, the team could eliminate some alternatives to simplify the trade-off analyses. The remaining options are due for more detailed analyses. A weighted scoring scheme could also be adopted to support the detailed analyses. The team assigns a weight to each criterion and rates each design alternative. Finally, the team ranks the alternatives by calculating a total score for each alternative. See [22] for detailed discussion. The approach provides a structured way for robustness or sensitivity analysis. Usually at this stage, the criteria that seem to differentiate the alternatives can be identified, which is critical in a trade-off analysis.

4.2. *Case Study 2*

This section briefly discusses another example where the approach has been applied. The example involved two subsystems of an AIN (Advanced Intelligent Network) product. The problem occurred for the deployment of new services. The development time was longer than planned. Two primary problems with the software architecture were identified. First, the Service Framework and the Platform were tightly coupled. As a result, the service designers, while developing new services or modifying existing services, often had to work on not only the Service Framework, but also the Platform. The second issue was that the service messages encoded with protocols were deciphered twice and the access of decoded messages by services was indirect. In other words, there was an extra level for both message decoding and accessing of the decoded messages by services.

Therefore, the three main objectives of this project are summarized as follows.

- Shorter time to market for new service deployment
- Better performance for service execution
- Lower coupling for evolution cost

We identified several alternatives and evaluated them against the reference architecture, i.e., the existing design. For brevity, the collection of metric data is not presented. Instead, Table 7 briefly summarizse the evaluation of three architectural alternatives.

5. Related Works

There is not much research reported in the literature on software architecture metrics. To the best of our knowledge, this article is an initial effort in developing a comprehensive list of architectural metrics and applying them to real projects based on a structured analytical methodology.

There are research efforts related to the evaluation of software architectural alternatives [13,15]. These two approaches share commonalities with ours. That is, they describe a structured method for making informed trade-off analysis. Making trade-offs is not a new concept. In fact, all design involves trade-off analysis. We also have borrowed the idea of concept selection from mechanical engineering [22].

Many trade-off analyses are performed in the "black-box" fashion. In this fashion, components usually are at a coarse level of abstraction or the components are treated as black-box. Evaluation, therefore, focuses more on the high level

Table 7. Evaluation of architectural alternatives selection criteria.

Selection criteria	**Alternative 1**	**Alternative 2**	**Alternative 3**
	• No change of message decoding, i.e., convert service messages to intermediate level, then to the internal target format. • Re-partition existing system components, functions. Decouple service-specific components from Platform.	• No change to message decoding. • Decouple service-specific components from Platform and allow services to directly access variables in shared memory block. • Dynamically allocate shared memory block tailored to suitable service size.	• Convert service messages to target format, no intermediate level.
Maintainability	+1	+2	+1
Portability	0	0	−1
Performance	0	+2	+1
Scalability	0	+1	0
Time to Market	−1	−2	−1
Backward compatibility	+2 (yes)	−2 (no)	0 (side effect on other parts)
Sum +'s	3	5	2
Sum 0's	3	1	2
Sum −'s	1	4	2
Net Score	2	1	0
Rank	1	2	3
Summary	• Less modification effort (4 components and 7 links). Effort required is small. • Backward compatible. • Problem occurs only at initialization/registration stage, but more verification needed. • The coupling problem between two subsystems is reduced.	• It would be much easier to maintain, but high modification effort is needed. • High performance gain and reduction of memory usage is expected, but quantitative measurement not available. • Long development time is needed. • Not backward compatible.	• Moderate modifications (2 components and 2 links). Effort is medium. • Performance gain not significant (∼2%) based on quantitative traffic simulation & measurement. • No backward compatability or service processing problem. But the intermediate format is also tied up with other traffic simulation.
Continue?	Yes • Quick solution, yet resolve the coupling problem between two subsystems. Hence, deployment of new services is faster.	No • Due to time-to-market and backward compatability. • May be long-term and larger scale option.	No • Not much gain in performance or other values. • Side effect on other parts of the system.

functional elements against a set of criteria, attributes, or high level requirements. Our approach goes beyond this step by conducting the analysis in the "white-box" manner. The level of abstraction coincides with the adopted metrics. This way, the analysis would be more informed and objective since there is more concrete information. Case study 1 shown in Sec. 4 is an example of "white-box" trade-off analysis. On the other hand, case Study 2, without the description of metrics collection and analysis, shows the trade-off analysis with the "black-box" flavor.

6. Lessons Learned

We have applied the approach to several projects. From the experience, we have some observations that are interesting and worth noting. First, there is other information that is beneficial to robustness analysis, particularly various architectural views and application-specific data. It is a generally accepted idea that multiple architectural views are needed [3,16]. The concept is also adopted in the recent UML (Unified Modeling Language) for object-oriented modeling and analysis. The architectural views used in our analysis include static view, dynamic view, resource view, and map view [17-19].

The resource view deals with the utilization of system resources, execution environments, and workload scenarios. Typical system resources are processors, memory, disks, and networks. The characteristics of the resources provide important information from the execution point of view. Various techniques can be used, including the identification of mapping of software onto hardware, performance modeling, measurements, parallel or concurrent processing, and simulation. Detailed discussion of the technique can be found in [19].

The map view is the principal difference between the approach and other techniques in architectural views. The map view is useful in determining allocation of functions to components or reducing system coupling. An example is illustrated in Fig. 3. It is obvious that the coupling is significantly reduced from (a) to (b). Coupling has a tremendous impact on many other quality attributes such as maintainability, portability, scalability, reusability, and interoperability [6]. Based on our experiences, sensitivity is usually highly correlated with coupling. This view is useful in providing information for architecture restructuring [20].

Second, historical application-specific data will also help us to study the relationships between high-level architectural components and the components at the code level in order to learn more on the robustness of software architectures, and to help support the management and cost estimation for future changes. Application-specific data include data on cost, defect, and code complexity. This type of information is necessary to translate the degree of effort required (High, Medium, and Low shown in Table 4) to product-specific cost adjustment factors.

Another often asked question about scenario-based analysis is "When do we stop generating scenarios?" [12]. Two approaches were used in our study in SEAL. First, scenario generation is closely tied to various types of objectives: stakeholder, architectural, and quality. Based on the objectives, we worked with domain experts closely and iteratively to identify scenarios and cluster these scenarios to make sure each objective is well covered. Much effort is spent in identifying the information early to help make informed trade-offs and decisions.

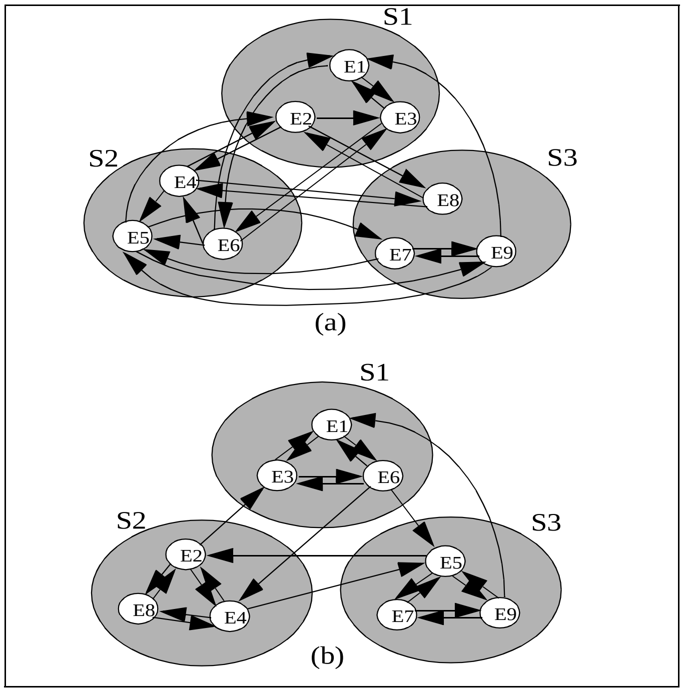

Fig. 3. An example of restructuring components to reduce coupling.

QFD (Quality Function Deployment) was then used to validate the balance of scenarios with respect to the objectives. A cascade of matrices are generated to show the relational strengths from stakeholder objectives, architectural objectives, to quality attributes [4]. Priorities are calculated for each objective. Finally, quality attributes are translated to scenarios to reveal the coverage of each quality attribute. An imbalance factor is then calculated for each quality attribute by dividing coverage by quality priority. If the imbalance factor is less than 1, we may need to develop more scenarios to address the quality attribute in accord with stakeholder, architectural, and quality importances. For instance, if the relative priority of performance is 18 and the coverage of performance by the scenarios is 9, the imbalance factor is 0.5. This suggests that more scenarios need to be developed to address performance.

Third, the area of of software architecture metrics is still in its infancy. Again, it is not our intention to identify a list of metrics that are useful to all applications. We do not even apply all the metrics listed in Sec. 3 to all the projects that we have worked on. Selection of the metrics is dependent on objectives, applications, and projects.

The final note is that the approach presented in this paper does not make complex decisions easy or completely objective. For some areas such as performance, it is easy to obtain objective and quantitative measurements. Some commonly used measurements include queries per second, execution time, and CPU utilization. On the other hand, there are still difficult decisions to make and there is still subjectiveness involved for other attributes.

Nevertheless, the value that this approach provides is a structured mechanism to support the trade-off analysis *early* and *explicitly*. By applying the method early, the potential problems can be solved at a lower cost. Having an explicit and structured method forces you to consider thoroughly about the objectives, alternatives, trade-offs, conflicts, and relevant resolutions or consequences in the decision process.

7. Summary and Future Research

This paper gave a concrete definition of software architecture robustness and a set of metrics to support robustness analyses. The Objective/Scenario/Metric paradigm was then introduced to support the measurement of sensitivity.

Sensitivity analysis is performed by going through the architecture with scenarios, identifying changes needed to meet stakeholder objectives, and classifying the changes according to the metrics definition. This paper also addressed change impact analysis from the perspectives of qualities, or architectural objectives and concerns. Finally, this paper discussed a way to analyze trade-offs among a set of design alternatives.

People use some methods for change impact analysis or trade-off analysis. However, this process is usually conducted implicitly or in an *ad-hoc* manner. The approach presented in this paper is explicit and in a structured fashion. The explicitness of the process can ease the capture and transfer of domain knowledge. The result could also reveal the potential reusability of a software product at the architecture level. Moreover, the approach facilitates the comparison between two architectures in the same problem domain from the evolution perspective. Two architectures in the same domain may have different structural alternatives and allocations from function into the structure. By explicitly classifying and identifying the types of changes, and performing the *delta analysis*, we can better estimate the effort required for different systems. The methodology was applied to the analysis of actual large-scale telecommunications software architectures.

At Nortel, we evaluate software architectures to identify quality aspects and architectural concerns for their evolution or for another project in the same problem domain. The metrics were extracted manually for this experiment. This process may be automated or partially automated for systems that are specified in architectural description languages (ADLs). Other metrics proposed for high-level design or object-oriented applications may be tailored and integrated into the methodology, and some of the existing metrics listed in this paper may be modified or removed as we learn more about them. We are currently pursuing research to apply Function Points to support cost and effort estimation.

An important factor that influences system evolution is the time interval allowed for making the changes. If the time allowed to make a certain amount of changes is short, the sensitivity usually is high. On the other hand, if the available period spans longer, the degree of impact becomes lower than that of a tight schedule. Timing aspect itself is a function of project planning, scheduling, revolution plan. We are working towards a mathematical definition of robustness by integrating various factors, including customer objectives, product attributes, nonfunctional qualities, and timing constraints. Further research needs to be conducted to quantitatively characterize software architectures and support trade-off analysis for evolution activities.

Acknowledgments

We are grateful to Sonia Bot and Rama Munikoti for their support.

References

1. V. Arora, Kalai Kalaichelvan, N. Goel, and R. Munikoti, “Measuring high-level design complexity of real-time object-oriented systems”, *Proc. Annual Oregon Workshop on Software Metrics*, 1995, pp. 91–94.
2. V. R. Basili, G. Caldiera, and H. D. Rombach, “The goal question metric approach”, *Encyclopedia of Software Engineering*, ed. J. Marcinak (Wiley, 1994).
3. L. Bass, P. Clements, and R. Kazman, *Software Archiecture in Practice* (Addison-Wesley, 1998).
4. S. Bot, C.-H. Lung, and M. Farrel, “A stakeholder-centric software architecture analysis approach”, *Proc. Int'l Software Architecture Workshop* (*ISAW*), 1996, pp. 152–154.
5. D. N. Card and R. L. Glass, *Measuring Software Design Quality* (Prentice-Hall, 1990).
6. H. Dhama, “Quantitative models of cohesion and coupling in software”, *J. Systems & Software* **29** (1995) 65–74.
7. D. Garlan and M. Shaw, “An introduction to software architecture”, *Advances in Sw. Eng. and Knowledge Eng.*, vol. 1, 1993.
8. G. Heyliger, “Coupling”, *Encyclopedia of Software Engineering*, ed. J. Marcinak (Wiley, 1994).
9. I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success* (ACM Press and Addison-Wesley, 1997).
10. K. Kalaichelvan and R. Munikoti, “Developing robust software architectures”, in *Workshop on Software Architecture Evaluation Best Practices* (*invited paper*), Software Engineering Institute, May 1996.
11. R. Kazman, L. Bass, G. Abowd, and M. Webb, “SAAM: A method for analyzing the Properties Software Architectures”, *Proc. ICSE* **16** (1994) 81–90.
12. R. Kazman, G. Abowd, L. Bass, and P. Clements, “Scenario-based analysis of software architecture”, *IEEE Software* **13**, 6 (1996) 47–55.
13. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, “The architecture tradeoff analysis method”, *Proc. 4th Int'l Conf. on Engineering of Complex*

Computer Systems, Aug. 1998, pp. 68–78.

14. T. Khoshkoftaar, E. Allen, K. Kalaichelvan, and N. Goel, "The impact of software evolution and reuse on software quality", *Empirical Software Engineering* **1**, no. 1 (1997) 31-44.
15. V. D. Kirova and H. G. Kradjel, "The DirSA case study: an introduction to software architecture technology", *Bells Labs Technical Journal*, July–Sep. 1998, pp. 125–137.
16. P. B. Kruchten, "The 4+1 view model of architecture", *IEEE Software*, Nov. 1995, pp. 42–50.
17. C.-H. Lung, "Empirical experiences in analyzing software architecture sensitivity", *Proc. COMPSAC*, 1997, pp. 164–165.
18. C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An approach to software architecture analysis for revolution and reusability," *Proc. CASCON*, 1997, pp. 144–154.
19. C.-H. Lung, A. Jalnapurkar, and A. El-Rayess, "Performance-oriented software architecture analysis: an experience report", *Proc. 1st Int'l Workshop on Software Performance* (*WOSP*), Oct. 1998, pp. 101-104.
20. C.-H. Lung, "Software architecture recovery and restructuring through clustering techniques", *Proc. 3rd Int'l Software Architecture Workshop* (*ISAW*), Nov. 1998, pp. 191–196.
21. D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture", *Software Engineering Notes* **17**, no. 4 (1992) 40–52.
22. K. T. Ulrich and S. D. Eppinger, *Product Design and Development* (McGraw-Hill, 1995).