Proc. of CASCON (Centre for Advanced Studies Conf.), Toronto, Canada, Oct. 1997, pp. 144-154.

An Approach to Software Architecture Analysis for Evolution and Reusability

Chung-Horng Lung, Sonia Bot, Kalai Kalaichelvan
Software Engineering Analysis Lab.
Nortel
Ottawa, Ontario, Canada K1Y 4H7
{lung | sdbot | kalai}@nortel.ca

Rick Kazman[1]
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
rnkazman@cgl.uwaterloo.ca

Abstract

Software evolution and reuse is more likely to receive higher payoff if high-level artifacts—such as architectures and designs—can be reused and can guide low-level component reuse. In practice, however, high-level artifacts are often not appropriately captured. This paper presents an approach to capturing and assessing software architectures for evolution and reuse. The approach consists of a framework for modeling various types of relevant information and a set of architectural views for re-engineering, analyzing, and comparing software architectures. We have applied this approach to large-scale telecommunications systems, where the approach is useful to reveal areas for improvement and the potential for reuse.

Keywords: Software architectures, product lines, analysis, software evolution, software reusability, scenarios

1. Introduction

Software evolution and reuse are two critical topics in industry, because of the huge expense involved and because of global competition. However, software systems are becoming increasingly complex, further complicating the already difficult problem of evolving or reusing software assets and products. To systematically support the process for the ever growing complexity of software, higher levels of abstraction are needed. Kruchten [9] noted that "for a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation" (p. 136). Software architectures are critical artifacts in bridging the gap between the initial concept of a system and the system's implementation; its low-level software components.

This paper presents a framework and a set of architectural views that were developed to assess software architectures for evolution and reuse built upon a scenario-based approach [7]. This framework is used to model different types of information, namely, stakeholder information [2,4], architecture information, quality information, and scenarios. Stakeholders can include, for example, designers, managers, and end-users. The information for stakeholders describes their objectives. Architecture information deals with the critical design principles or architectural objectives. Quality information refers to the non-functional attributes such as performance, modifiability, availability, and integrability. Scenarios are narratives that describe use cases of a system. Scenarios can be used to capture the system's functionality. Scenarios that are *not* directly supported by the current system can be used to detect possible flaws or to assess the architecture's support for potential enhancements. A set of scenarios is derived from the stakeholder objectives, architectural objectives, and desired system quality attributes or objectives. Section 2 will give a more detailed discussion on this topic.

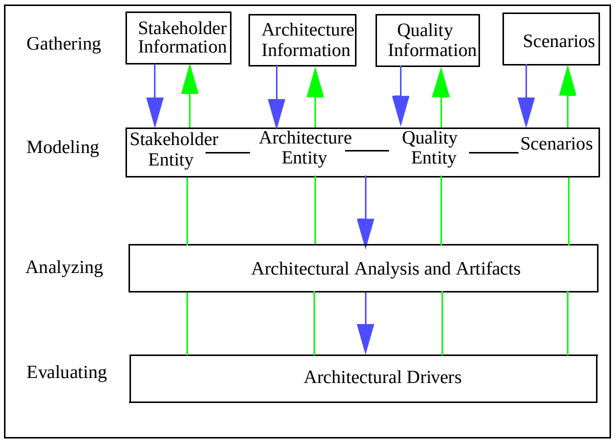
Objectives provide boundaries and drive the analysis. Architectural views are important for evolution and reuse, because various views provide different perspectives, which are useful in understanding, re-engineering, and analyzing systems. In addition, these architectural views support analysis of systems developed using different paradigms. For example, one application that we have made of the

1. Kazman's current address: Software Engineering Institute, Carnegie Mellon Univ., Pittsburgh, PA, 15213. E-mail: kazman@sei.cmu.edu

views is to compare systems developed using functional decomposition and object-oriented design in the same problem area.

The main objective of the approach was to assess an existing architecture for project evolution or reuse in a future project in the same problem domain or product line. The work reflects empirical experience gathered by an external review team to evaluate the sensitivity of an architecture to changes in key customer value parameters. An example of a customer value is scalability. For instance: what is the sensitivity of an architecture if the system is to be modified so that it supports fifteen features, instead of the previous ten, and at the same time the system is to be scaled from processing fifty calls to eighty calls per minute?

FIGURE 1.
Framework for Architecture Information Gathering and Analysis



The remainder of this paper is organized as follows: Section 2 demonstrates the framework for analysis. Section 3 describes the context of architectural views and various architectural views adopted for the analysis of software architectures. Examples of the views are also demonstrated in this section. Section 4 highlights some example scenarios and partial analysis results. Section 5 presents some important lessons learned from applying the approach to several telecommunications systems. Finally, Section 6 gives our concluding remarks.

2. Framework for Information Gathering and Analysis

To ensure that the software architecture analysis process is organized and scientific (and hence, repeatable), a framework for architecture information gathering and analysis was formulated, as described in Figure 1. The activities described in the framework are performed iteratively instead of in a strict sequential manner.

Gathering. This phase focuses on becoming aware of the available and required information to do the analysis, and then to collect and compile it. Currently four categories of information are being addressed: stakeholder, architecture, quality, and scenarios or use cases. In the future, the information categories may be extended, to include for example, competitive analysis.

Modeling. Once it is gathered, the information is then aligned across information categories. The focus here is on mapping stakeholder, architecture, quality, and scenario information into usable artifacts. This information is used to direct the capture of the architecture (if it is not already recorded in a usable form) and to drive the analysis. It is also a critical vehicle in providing feedback in the latter phases.

Modeling is a critical phase, since if it is not done correctly, it can mislead and skew the rest of the analysis. In the modeling phase both the breadth and depth of the analysis are taken into account. The breadth aspect describes the relationships between: stakeholders objectives, architectural objectives, quality attributes, and scenarios. For example, it is useful to form a matrix of quality attributes and stakeholders, to ensure that each attribute is at least considered from the perspective of each stakeholder.

The depth aspect deals with the levels of abstraction at which the stakeholder objectives are represented (and hence analyzed). A single stakeholder objective or an architectural objective could be represented by several quality attributes or scenarios, each describing one aspect of the objective. The depth at which various types of information are represented will affect the accuracy (and cost) of the analysis. Modeling of the depth aspect is supported by adopting software QFD (quality function deployment) [2], where relational matrices are used to prioritize high-level objectives and the results are fed into corresponding objectives at the next level.

Analyzing. This phase focuses on specific software architecture analysis and generation of artifacts to do the analysis. Examples of artifacts include: domain models (which help in comparing competing architectures within the same functional area [6]); relevant architectural views; scenarios; environmental assumptions and constraints; and tradeoff rationale. SAAM (Software Architecture Analysis Method) [7] is adopted and extended for the analysis. Explicit scenarios are mapped onto an architecture for analysis of quality attributes.

Evaluating. This phase focuses on drivers for architectural development. In this phase recommendations are made, “hot spots” in the architecture (areas of high predicted complexity, large numbers of changes, performance bottlenecks, etc.) are located and strategies for their mitigation are enumerated, common reference models (independent of architecture capture) are identified. It is important that this phase ties back to the stakeholders’ values, as they are the drivers of the analysis in the first place.

2.1 Example of Modeling of Objectives

Having described the framework, we now give an example in the domain of telecommunication switching software. In this example we show a couple of stakeholder objectives, architectural objectives, and quality objectives, and the alignment of these three types of objective in Table 1.

Table 1: Stakeholder-Architectural-Quality Objectives: An Example

Stakeholder Objectives	Architectural Objectives	Quality Attributes
Allow interworking with other products and third. parties	Expose functionality which provides the implementation of standardized third party application programming interfaces.	Reliability Modifiability Portability
Allow independent development and incremental delivery of new features.	Decouple functionalities and use of virtual interfaces.	Reliability Modifiability Integrability

A set of scenarios are then developed based on the stakeholder and architectural objectives. Each objective may consist of a set of scenarios or scenario classes. Each scenario class in turn consists of various number of scenarios or sub-classes. An example of scenarios will be presented in Section 4. The objectives also are important factors in determining when to stop generating more scenarios. This concept will be addressed in Section 4 as well.

3. Architectural Views for Evolution and Reusability Analysis

The development of a complex software system involves various stakeholders. Diverse stakeholders have different needs and perspectives of the system. Each perspective represents a partial description of a system. A complete description of a system requires multiple viewpoints. In addition, various viewpoints may be needed at various stages in the life cycle. An architectural view, in this context, is a perspective that satisfies the expressed needs of a stakeholder.

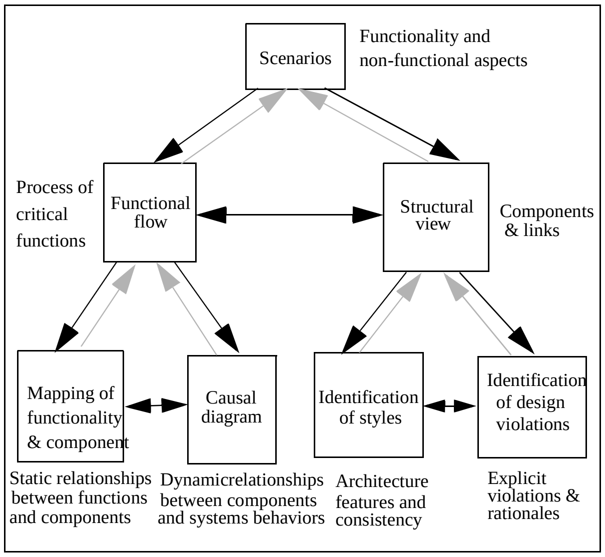
SEAL has adopted various architectural views that are critical for software architecture analysis. The set of views includes: a static view, a map view, a dynamic view, and a resource view [11]. Each view and some commonly used methods are briefly described below.

- **Static view**. The static view shows the overall topology. The methods that can be used for this view include logical diagram, structure diagram, object diagram, and module diagram.
- **Map view**. The map view identifies the style, design violations, and the mapping between components and functions or features. An example will be presented in the next section.
- **Dynamic view**. The dynamic view addresses the behavioral aspects of a system. This view can be supported by functional or operational diagram, causal diagram, messaging diagram or message sequence chart, object interaction diagram, state machine, and Petri net.
- **Resource view**. The resource view deals with the utilization aspect of the system resources. Various techniques have been used in support of this view, including the identification of the mapping of software onto hardware, queuing model, simulation and performance.

The development of the views does not have be carried out in a strict sequential manner. Rather, the process is iterative in nature. Further, not all the views may be needed for each evaluation and each view is not constrained by a particular method or notation. Selection of appropriate views and suitable methods depend on the specific application environment and stakeholder values.

Figure 2 demonstrates a real usage of these views for a project. The structural view corresponds to the static view. The functional flow and the causal diagram belong to the dynamic view. The map view consists of three items as just described. The resource view was not incorporated for this exercise primarily because the main objective focused on evolution and reuse perspective, and a separate team was working on the performance issues. The views and their relationships are described next.

FIGURE 2. An Example of the Usage of Architectural Views


Scenarios
Functionality and non-functional aspects
Process of critical functions
Functional flow
Structural view
Components & links
Mapping of functionality & component
Causal diagram
Identification of styles
Identification of design violations
Static relationships between functions and components
Dynamicrelationships between components and systems behaviors
Architecture features and consistency
Explicit violations & rationales

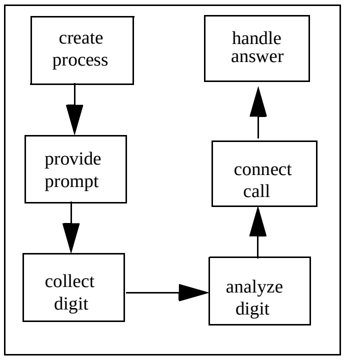
Scenarios. In this study, scenarios are the main driver for the capture of other architectural views and for the analysis of an architecture. To begin the analysis process, a few scenarios are typically selected to identify and understand the system's critical functionality.

Functional Flow. The functional (sometimes called *operational*) flow, in this context, refers to the sequence of functions that are identified based on a set of scenarios. This view reveals how the system works to realize particular scenarios.

Most architectural representations emphasize only *static* entities: the system's "boxes" (components) and "links" (connectors). A high-level functional flow view aids understanding by showing the critical system functions and the processing of these functions: an *operational* view of the system. This view is simple but useful. In our experience it is particularly useful for an object-oriented system where frequently only the modelled real-world entities are described rather than showing how the system actually functions.

To return to our telecommunications example, there are large number of features in an advanced telecommunications system. Understanding the system as a whole is an enormous and daunting task. So, a typical scenario for beginning to understand such a system would be to model a normal telephone call. A simplified functional flow for a normal phone call in a call processing system is shown in Figure 3.

FIGURE 3. Functional View for a Hypothetical System

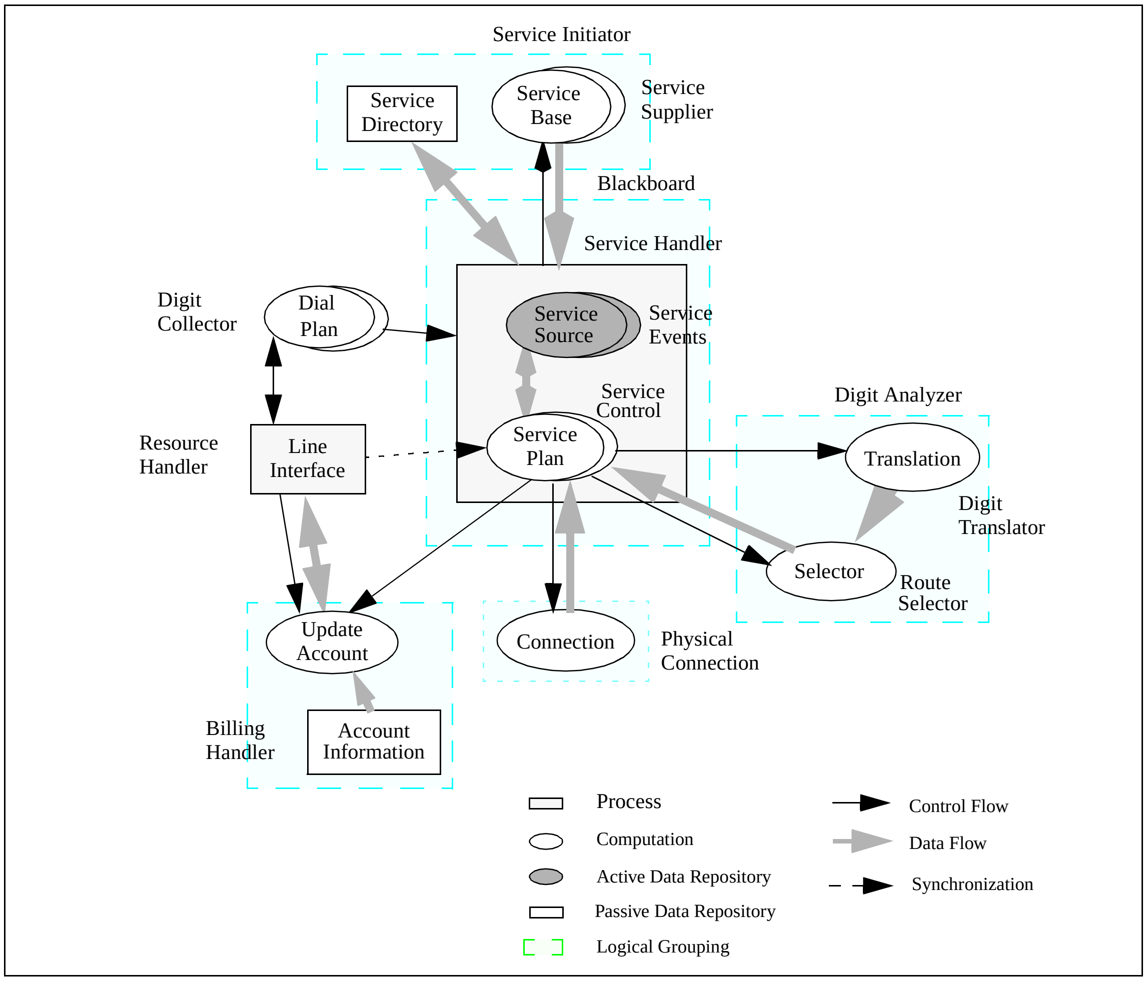


Structural View. Existing legacy systems usually do not have appropriate pre-existing architectural representations. Consequently, to analyze a software architecture, a representation is needed that shows the overall system topology. This view integrates and extends two methods presented in [5] and [6] to address the classification and generalization of a system's components and functions, and the connections between components.

The classification and generalization of components and connections also facilitates the estimation of cost or effort required for changes to be made. For instance, the cost for a change to be made to a processing unit normally would be higher than a change to be made to a data repository. Such early (and *intentionally* crude) estimates help in determining where to place more effort in an architectural analysis.

Mapping between Functions and Components. The mapping between functions and components provides a view that supports traceability analysis,

FIGURE 4. Structural View for a Hypothetical Call Processing System



especially if there is any modification to be made to the system. Two different representations are used for the mapping. Table 2 shows the mapping of the system's main functions or features to components, whereas Table 3 demonstrates an example of the mapping of components to functions or features. The components involved are tied back to those shown in the structural view as shown in Figure 4. The table helps locate all components involved for a particular function. The book-keeping effort in creating and maintaining such views, and the links between them, is crucial to supporting analysis. Humans can not be expected to keep all the details in their heads, all the time.

Table 2: Mapping of Functions to Components: An Example

Function	Components Involved
Digit Collection	Dial Plan, Line Interface, Service Handler, Service Initiator
Call Connection	Service Handler, Line Interface, Billing Handler
Answer Handling	Service Handler, Line Interface, Connection

Table 3: Mapping of Components to Functions: An Example

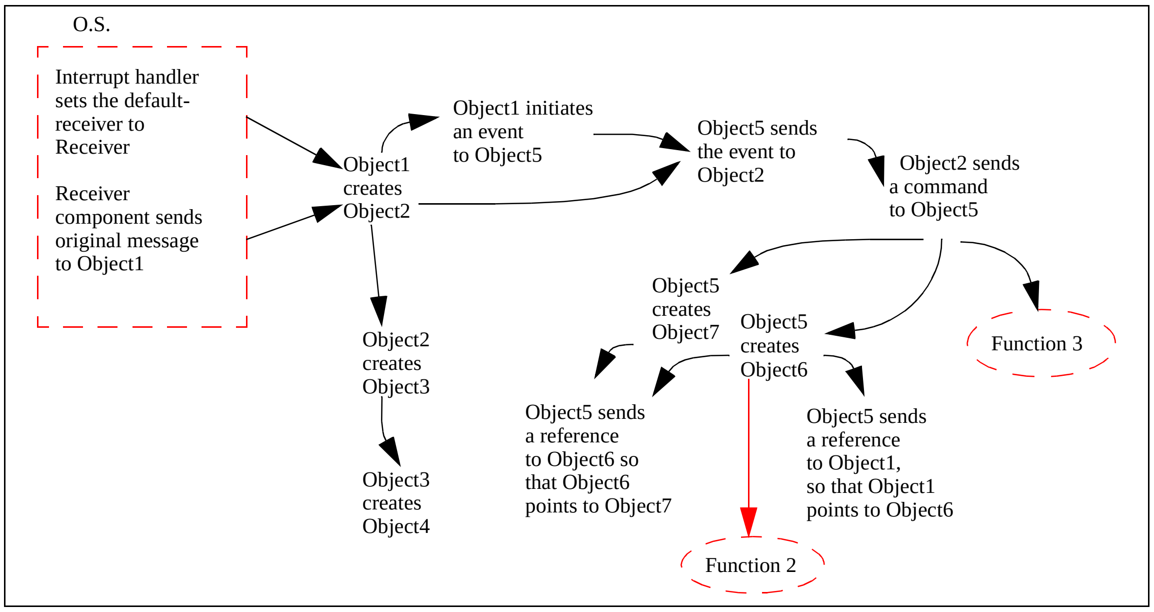
Component	Functions Involved
Dial Plan	Digit Collection
Service Handler	Digit Collection, Call Connection, Answer Handling
Line Interface	Digit Collection, Call Connection, Answer Handling

Table 3 shows the mapping of components to functions for a particular scenario. The mapping supports the identification of the functions that a component contributes. The functions identified for the mapping do not have to be specific to a system. In other words, these functions could also be generic to a application area such as a set of reference functions for the purpose of comparing different systems. When sets of functions are broadly agreed upon and re-used, we have a reference model.

The tables, though conceptually simple, are useful in demonstrating different aspects of functions and components. The concept is similar to spreadsheet software where diverse representations can be quickly generated based on a user's needs. The tables can also be used as a quick-and-dirty analysis of functional cohesion and coupling. If a function involves too many components, this function may need to be decomposed further into several sub-functions. In addition, the information could be used to cluster components based on the cooperations and dependencies of components. For instance, the components Service Handler and Line Interface presented in Table 3 show higher functional cohesion as both components are related to a set of common functions.

Causal Diagram. Architectural representations most commonly describe static features, things like: components, the relationship between components, high-level functionality, and allocation to hardware. The *behavioral* aspect of the system is important for high-level understanding, communication among stakeholders, architecture evolution, and re-engineering. This view also supports the development of an accurate static view and helps validate the consistency of the other representations.

FIGURE 5. Dynamic View: An Illustration of "Create Process" for the Hypothetical Call Processing System


O.S.
Interrupt handler sets the default-receiver to Receiver
Receiver component sends original message to Object1
Object1 creates Object2
Object1 initiates an event to Object5
Object5 sends the event to Object2
Object2 sends a command to Object5
Object2 creates Object3
Object3 creates Object4
Object5 creates Object7
Object5 creates Object6
Object5 sends a reference to Object6 so that Object6 points to Object7
Object5 sends a reference to Object1, so that Object1 points to Object6
Function 2
Function 3

Various methods could be used to model the behavioral aspect of a system. Examples include state machines, message sequence charts, and Petri nets. A generic causal representation is presented in Figure 5 as an illustration [10]. The tail of an arrow reveals the cause, while the head of an arrow depicts the effect. For each function in the functional flow, there is a corresponding causal diagram to reveal the behavioral aspect. Figure 5 is an example of "create process" demonstrated in the functional flow (as shown in Figure 3.)

The behavioral aspect is important to understand the system before reuse occurs. In addition, the dynamic view also supports maintainability as a system evolves. For instance, if modifications are made the static architectural representations may stay the same, but some of the system's behaviors may be modified. The modification of behaviors should be, but typically can not be, explicitly represented by static architectural views. Another example is that if personnel changes or the architect leaves, there may be different interpretations for the static view by other designers or new employees.

Identification of Architectural Styles. An architecture can be classified into more than one style and an architecture allows coexistence of multiple styles [5,8]. The primary purposes of the style or pattern is to impose an overall structural interpretation on a software system or subsystem for consistency checking, and to support human to human communications of the software.

For the example shown in Figure 4, the behavior of the architecture is similar to a blackboard [3,5], since the system has a centralized control, called a service handler, to coordinate a group of components. The identification of an architectural style help focus on critical features such as the control mechanism of a style, the communication mechanism between components, and the integrability of new component, or the modifiability of existing components. These important features for the blackboard model are identified for more detailed analyses as listed in Table 4.

Table 1: Features to Focus on for the Analysis of the Blackboard Model

Control/ Registration mechanism	• When the blackboard wants to send a message to some units, does it broadcast the message to all the units or simply send the message to the registered units? • Does the model support independent control or broadcast control? • Is the control single-threaded or multi-threaded? • Is the message control, data, or both?
Communicatio n mechanism	• Is there a specific point of contact or multiple points of contact between the blackboard and the computational units?
Violations	• Are there any links that violate the control or communication policy?
Integrability and modifiability	• If new components are added to the system, will they be integrated into the blackboard the same way as existing components?

In addition, the analysis can support the decision-making process in choosing an appropriate style for the target domain or trade-off analysis. The appropriate style can then be reused for the target domain, even if the architecture itself is evaluated to be risky to be directly reused for the target. For large systems where multiple styles may exist, analysis of style interoperability is important. Style interoperability is directly related to system integritymaintainability. It is important to identify and analyze how one particular style communicates with other styles [1].

Identification of Design Violations. This view deals with the components or links that are missing or are not represented properly, and the control or communication mechanisms that violate the policy of the identified architectural style. The architectural style may only reveal an "idealized" or "as-intended" software architecture initially developed by a group of software designers. This view, on the other hand, recovers the "as-built" aspect of an architecture supported by the causal representations. For instance, the blackboard's control mechanism requires a single point of contact between the central control unit and the other cooperative components, but the architecture that follows the style, in fact, has multiple points of contact under certain cir-

circumstances.

Some reasons for the violations could be legacy systems, modifications for performance, understandability, and discrepancies in the levels of abstraction. The violations must be explicitly documented to reduce potential problems caused by ambiguity or inconsistency. The documentation can also support system maintainability. Architectural violations are as important as normal architectural features and must be identified before reuse occurs to reduce unnecessary maintenance effort.

4. Examples of Scenarios and Analyses

To make a concrete evaluation for the architecture, a number of explicit scenarios are developed based on 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.

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.

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 are not and 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.

The following highlights a couple of scenarios and partial analysis results for the objectives shown in Table 1. The analysis is based on the hypothetical architecture depicted in Figure 4.

<u>Scenario 1</u>: A third party develops a new feature to interwork with the architecture.

<u>Architecture Impact</u>: Interfaces for third party have not been implemented. Proxies are needed to communicate with third party applications. Further, new features need to added to the service source and service plan shown in the structural view in Figure 4. More explicit information on new features need to be identified for further analysis, however.

<u>Scenario 2</u>: The system will be delivered with basic capabilities. New features for complex call processing will be incrementally introduced.

<u>Architecture Impact</u>: The architecture supports incremental development because of the separation of concerns, decoupling of functionality through the blackboard, the controlled mechanism for service interactions, and a mechanism used specifically for incremental delivery. Further analysis on performance and memory capacity needs to be conducted.

Scenarios could be described in different levels of detail. Based on the stakeholder objectives and preliminary analysis results, some scenarios may be further refined or other scenarios in the same category may need to be developed. For an application, just a few scenarios were initially developed collaboratively with the architect. After the analysis and discussion with the architect, a lot more scenarios were generated for further evaluation.

5. Lessons Learned

We have applied this framework and set of views to several projects within Nortel. The analysis is heavily based on stakeholder objectives. For example, in one project we grouped the stakeholder objectives into five categories and added additional two to cover as many areas as possible. One was for basic needs, the other one was for potential future changes that were not described in the stakeholder objectives. Over thirty scenarios were then developed and classified based on the objectives for this exercise. For another much smaller project, we ended up with more scenarios than the previous example for deeper analysis.

We adopted and extended SAAM [7] by not only identifying, for each scenario, required changes, but also estimating the effort required (low, medium, or high) to make the changes based on the required changes and domain experts experiences. These two types of information together gave us a better idea of how the system could support each of the objectives or the risk levels for system evolution or reuse across applications than just counting the number of changes.

Further, the analysis could qualitatively reveal the reusability aspect of an architecture. By identifying and analyzing areas that are reusable, tailorable, or not reusable based on explicit scenarios and various insight views, rather than design from scratch, the development time for the architecture and high-level design for a new one project in the same product line was reduced. For instance, the service handler and service initiator in Figure 4 are highly reusable, and are easy to modify or enhance based on the current control and communication mechanisms. On the contrary, the risk level of reusing the existing resource handler shown in Figure 4 could be high due to its idiosyncratic implementation. Similar results were obtained for a real project, where parts of the architecture got reused and some areas were overhauled for a new project.

Three different tabular representations are also used to summarize the results. One representation shows the analysis results based on objectives. A summary is also attached for each objective to address identified changes and overall effort required or risk level involved for the required changes for evolution, or suitability of the architecture for another project. The second representation demonstrates scenario interactions. For each component, the list of scenarios that cause changes to it are listed. The third representation is a summary based on quality attributes. Similar to the previous representation, the scenarios that have significant impact on the qualities are listed. We found these representations highly useful devices for communicating with stakeholders.

The role of views. The scenarios are the main drivers to evaluate various areas of an architecture. The architectural views can reveal deeper information, however. Scenarios describe important functionality that the system must support or identifies where the system may need to be changed over time. Scenarios and the structural view are effective in identifying components that need to be modified. From the maintenance perspective, scenarios are useful for adaptive and preventive maintenance activities [13], but are less effective in corrective and perfective maintenance activities. Other architectural views must be used to support the analysis.

For instance, analysis of scenario interaction is a critical step in SAAM. A high degree of scenario interaction may indicate that a component is poorly isolated [7]. However, the style view may show that this is just the nature of a particular architectural pattern. For instance, the blackboard in the blackboard model highly interacts with other components. In this case, the focus is shifted from scenario interaction to consistency checking of the architecture and its style. The dynamic view may then be appropriate to examine the behavioral aspect to validate that the control and communication are handled in an expected manner. Another example is that an identified violation or shortcut in the existing system for performance purposes may not be needed in the future if the system is ported to a faster platform. Another possible reason for violations could be legacy systems. A project that we dealt with overhauled a legacy system. In this case, some known violations were not carried into the new design. Hence, the maintainability of the system could be improved by removing the violation. Violations were also used to validate the conformance of the implementation to the architecture. Similarly, the mapping between components and functions can reveal the cohesion and coupling aspects of the system. This view is useful for system partitioning and maintenance, especially for “ripple effect” analysis.

Scenario generation. Another often asked question about scenario-based analysis is “When to stop generating scenarios?” [7]. Two approaches were used in our study in SEAL. First, scenario generation is closely tied to various types of objectives: stakeholder, architectural, and quality. We spent a lot of effort in identifying the information up front. 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.

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 [2]. 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.

6. Summary

This paper presented a framework and a set of architectural views for the analysis of software architecture for evolution and reusability. The approach was developed from empirical studies on large-scale telecommunications systems for the assessment of reuse across applications and for system evolution. The scenarios are aligned with stakeholder objectives, architectural objectives, and quality attributes. The scenarios can also be reused across applications. More importantly, the analysis reveals the sensitivity of a system due to the change in or the importance of objectives, and future requirements.

The method also could facilitate the comparison of different architectures developed in the same domain using different paradigms (e.g. OO vs. functional decomposition) by using concrete scenarios aligned with the other views. In Section 5, analysis results for one architecture were illustrated. Should another architecture developed in different paradigm be in place for comparison, the comparison would be performed by identifying components that need to be modified, added, or removed based on scenarios mapped onto the other architectural views. The effort required to make the modifications for different architectures could also be estimated based on complexity information, architectural views, and historical data for comparison.

Due to proprietary reasons, detailed architectural and analysis results could not be presented. In SEAL, we have used this technique to analyze a system for better understanding and project evolution. The technique was also used to compare two complex call processing systems with respect to their fitness for a new project. The critical successes of using the technique included better understanding of target systems, better communications among various stakeholders, identification of development of reusable assets, and extraction of problem areas or sites of complexity. Furthermore, the technique even helped the senior designers better understand architectural issues in their own systems. The capture of architectural views and mapping of various objectives also were useful information for existing systems, especially personnel changes are practical and training for new employee is important.

In SEAL, we have other teams that are working on the complexity measurement of high-level design and code. This measurement provides insights of complicated components for detailed analysis and more accurate estimation of the effort required for the changes. In other words, life cycle end-to-end analysis is supported for various software products. We are also developing and validating of a set of metrics for quantitative assessment of software architectures [12].

References

[1] D. Belanger, et al., Architecture Styles and Services: An Experiment Involving the Signal Operations Platforms-Provisioning Operations System, *AT&T Technical Journal*, Jan/Feb 1996, pp. 54-63.

[2] S. Bot, C.-H. Lung, and M. Farrell, A Stakeholder-Centric Software Architecture Analysis Approach, in *Proc. ISAW 2 - Int'l Software Architecture Workshop*, 1996.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[4] C. Gacek, A. Abd-Allah, B. Clark, B. Boehm. On the Definition of Software System Architecture, in *Proc. of ICSE 17 Software Architecture Workshop*, April 1995.

[5] D. Garlan and M. Shaw. An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, vol. 1, 1993.

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

[7] R. Kazman, G. Abowd, L. Bass, P. Clements. Scenario-Based Analysis of Software Architecture, *IEEE Software*, Nov 1996.

[8] P. B. Kruchten. The 4+1 View Model of Architecture, *IEEE Software*, Nov 1995, pp. 42-50.

[9] C. Krueger, Software Reuse, *ACM Computing Surveys*, 24(2), 1992, pp. 131-183.

[10] C.-H. Lung and J. Urban. An Expanded View of Domain Modeling for Software Analogy. *Proc. 19th Annual Int'l Comp Software & Applications Conf - COMPSAC*, pp.77-82, 1995.

[11] C.-H. Lung, Empirical Experiences in Analyzing Software Architecture Sensitivity, in *Proc. of COMPSAC*, pp. 164-165, 1997.

[12] C.-H. Lung and K. Kalaichelvan, Metrics for Software Architecture Robustness Analysis, *submitted for publication*.

[13] S. Wage, Preventive Software Maintenance: Prevention is Better Than Cure, *Tech. Report*, School of Info. Science and Technology, Liverpool Polytechnic, 1988.