Reflection on Software Architecture Practices – What Works, What Remains to Be Seen, and What Are the Gaps

Chung-Horng Lung
Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
chlung@sce.carleton.ca

Marzia Zaman, Nishith Goel
Cistel Technology
Ottawa, Ontario, Canada
{marzia, ngoel}@cistel.com

Abstract

This report presents a reflection on software architecture practices based on our past ten year's industrial experiences, particularly in the areas of communications networks and telecommunications and. The report summarizes the methods, tools, and techniques that we have used on various projects. We also discuss, based on our experiences, what methods are useful, what remains to be validated, and what the gaps are between the state of practices and our desired wishes.

I. Background and Past Experiences

We started the software architecture analysis program in the Software Engineering Analysis Lab (SEAL) at Nortel Networks in 1995. Since then, we have developed or experimented some methods or techniques to help us conduct software architecture analysis or support architecture evolution of products developed within the company. In some cases, the analysis was conducted after the product had been built and was mainly used for evaluation purpose. While in some cases, we applied some methods during the process of design or evolution. The result of the analysis facilitated the design or re-engineering of the product.

The program ended in 2001 at Nortel. After that, we have been continuing on some research and practices on some third party systems. This report briefly describes those methods and their effects based on our experiences. We also identify some areas that could be potentially useful on the state of practice of software architecture.

II. Methods Used and What Works and What …

We have adopted, adapted, and developed some methods and techniques to support software architecture analysis. The following is a list of those methods and the applications of them. We also briefly describe the effects of those methods based on our experiences, primarily in the area of competitive telecommunications.

1. Development of measurement technique for real-time object-oriented systems based on high-level design using ObjecTime graphical tool [Arora95]. The metrics were obtained by parsing the textual version of the design to identify complex components or subsystems. The main challenges were the understanding or interpretation of the metrics and the maturity of the graphical tool. Many metrics are still not well understood. Changes had to be made when the internal representation of the graphical tool was modified, which was time consuming and an unpleasant process.

2. Adaptation of the stakeholders-centric [Bot96] approach based on Gacek, et al., [Gacek95] and extension of the scenario-based software architecture analysis method from SAAM [Kazman95, Kazman96]. During the process, we also identified issues related to scenario coverage. In other words, when should we stop generating more scenarios? QFD (Quality Function Deployment) was adopted to trace the priority between the stakeholder objectives and corresponding architectural / quality objectives and scenarios. The results indicated that twice as many scenarios were needed for particular classes, which was usually conducted in an ad-hoc manner or ignored [Bot96]. QFD emphasizes on the customer needs and could provide a balanced view in terms of scenario coverage. However, in practice, in highly competitive areas, time-to-market may dominate in the initial phase or requirements could not be clearly identified even by customers. In such cases, usage of QFD becomes less compelling.

   Stakeholders and scenarios have been widely appreciated as necessary ingredients in software architecture analysis. QFD provides useful feedbacks and balanced views in terms of scenario generation and coverage. But the approach does not address the issue of concurrent scenarios. We also have adopted design combinatorial theory that has been discussed intensively in testing [Lung05a] to support concurrent-scenario coverage. The motivation is that a system typically supports multiple functionalities at the same

time. Those functionalities may work correctly when executed individually, but may encounter faults when run collaboratively, especially for complex concurrent or distributed systems. Design combinatorial theory from the testing discipline is useful to identify potential complicated scenario interactions. But it has not been verified yet in terms of acceptance.

Take a case study as an example [Lung98a]. The system behaved well while three scenarios were conducted separately. However, performance degraded significantly when these three scenarios happened simultaneously. The use case where these three scenarios could happen at the same time was initially missed, which caused performance problem in a real-time telecommunications system. Using a more formal approach reduces the risk of missing use cases where many scenarios may occur simultaneously.

3. Various views are advocated to support architecture design and evaluation [Kruchen95]. We also developed and evolved several architectural views to meet our needs [Lung97, Lung00]. The concept has also been well accepted and built into UML. Various views provide diverse aspects. However, in practice, not all the views depicted in UML may be needed or some other view(s) may be needed for particular cases or stakeholders. For instance, a more general notation is more useful to communicate with product requirements teams or business analysts. Another issue with UML is that architecture design frequently changes for a new project. Architects or designers typically do not capture various diagrams in details in forward engineering, except the basic needs, due to fact that they will be changed anyway. Maintaining the consistency between views is time consuming, tedious, and volatile due to frequent changes. Typically, only lightweight modeling in the early stage is needed to avoid over-diagramming. This point has also been addressed in the Unified Process.

4. Identification or capture of architecture styles or patterns can facilitate communications between stakeholders, especially designers. However, some designers may not be familiar with some technical terms, even though they have used something conceptually identical or similar for some time. This brings to the social aspect of this field. Senior architects or designers may feel uncomfortable if "advanced technologies" are brought into managements attentions.

   Styles may be too abstract. For example, layered architecture is standard in network communications. Everything evolves around the style. Additional insights are expected. Patterns, on the other hand, capture more specific solutions. But patterns may not be that easy to understand if they are new concept. Concrete examples and case studies using patterns are extremely helpful for the designers [Lung02b]. The concept of case-based reasoning and software analogy could be tied together with patterns to provide more leverage.

5. Software architecture recovery and reengineering is often inevitable due to changes in requirements and evolution. Tools (code browsers, reverse engineering tools) are useful to recover the software architecture or design. One of the techniques that has been intensively studied and discussed as a software reengineering technique in the literature is software clustering. We have been applying clustering techniques to many projects in this area. For some cases, just capturing the architecture has tremendous value. For others, much more detailed analysis is needed to identify specific problem areas or bottlenecks for improvement, e.g., software performance engineering [Smith90] or architecture reengineering [Lung98b, Lung04, Kostelijk05].

   Software performance is a tricky issue in architecture evaluation. It requires a lot of details, which often may not be supported at the architecture level. Performance modeling can be useful in this phase [Smith90, Lung98a]. However, there are issues related to performance modeling. First, modeling itself may take a long time. In one exercise of a telecommunications system evaluation, model building along took several months, which is generally not acceptable in a highly competitive industry. Second, there are constraints or limitations for the modeling techniques. In another real case study, the results from modeling were very different from the actual measurements of the reengineered implementation. In addition, software performance engineering (SPE) is a specialized area, which people may not be familiar with. Academic researchers usually adopt modeling techniques, whereas industrial practitioners mostly rely on measurements. Modeling is valuable to help us better understand the system and is better than "build-break-fix" approach. However, in some practical areas, we need to consider more detailed information than just that of typical high level modeling techniques to provide more useful insights.

   Software architecture visualization is another useful approach to display the system and components and the relationships of components for an overall understanding of the system for maintenance and/or evolution. Architecture visualization can be coupled

with other information such as software metrics to show the complexity, performance and reliability [Zaman99]. One caveat is that a powerful visualization tool is expensive. Many organizations are reluctant to invest thousands of dollars on something that is not so critical to the project development.

6. ATA (Architecture Tradeoff Analysis) [Kazman99] or sensitivity analysis based on some architecture metrics [Lung00] or high-level design metrics [Arora95] is useful for comparisons, especially in cases where multiple candidate architectures are possible. Those approaches can help identify more specific areas that are critical or different. Quantitative measurements may be needed for some qualities, e.g., performance. But quantitative analysis may not be available. Qualitative reasoning sometimes is the only option, but ill-founded reasoning can favor unsupported tradeoffs [Kostelijk05]. Unfortunately, this is a reality at this level due to uncertainties of technologies and/or requirements, and timing pressure. Even if quantitative data are available, it dose not mean that they are well understood for some qualities. For instance, what does it mean it you can get a final value of X for maintainability? In some cases, the most critical attribute dominates, even if the final score, if a scoring mechanism is adopted, may be lower than other options.

   There are still challenges in this area. On one hand, we need concrete evidence to demonstrate the benefits of one alternative over the other. On the other hand, it is often difficult to derive concrete results at this level, particularly for new systems. A typical example is "How to evaluate the performance or availability at the architecture level?" In telecommunications, availability is crucial and the industrial requirement for this quality is 99.999%. But how to support the claim or compare multiple alternatives at this level? The traditional model used in telecommunications is passive replication, e.g., one active and one standby processor with a switch to connect the clients to the current active processor. Another model is active replication, e.g., clients talk to a group of servers, but only one will respond. But there are many possible alternatives in between [Yu00, Hobbs05].

   Evaluation of an alternative could be very complicated and time consuming. Modeling techniques can provide useful information in this case, but they require predicated rates (e.g., failure rates or performance data) of components that are as close to the real system as possible. For hardware components, the rates have been systematically captured based on testing and real results from the field and are published. But the relevant data generally are not available for software components, which is a great challenge in software architecture evaluation. Moreover, how to support the architect to make the decision even given the modeling results is another challenge. For instance, if the availability modeling result is 99.99% (which is lower than the *carrier grade* product requirement of 99.999%) does it mean that we should find an alternative architectural solution or continue the design and tune the product at the end? Increasing the value from 99.99% to 99.999% of a product may require tremendous efforts in practice. On the other hand, if the result obtained from the modeling is better than five 9s, does it guarantee that the product will satisfy the requirement?

7. From requirements to architecture has been discussed intensively, e.g., [STRAW01, STRAW02]. We have experimented methods, based on the idea postulated by Alexander [Alexander64] for system decomposition. Decomposition provides heuristics on how to partition the system into subsystems to increase cohesion and reduce coupling. We adopted the method to identify the relationships between requirements and cluster requirements based on the relationships [Lung02a] on a new system in network traffic controller. Unfortunately, it turned out that it was difficult to identify the relationships between requirements, because requirements may not be clear or may be ambitious, or the level of abstraction may be different for various requirements. This is particularly true for a new system.

   We have modified the approach by identifying the relationships between requirements and a set of relevant attributes instead. After that, we used the clustering technique to group related requirements to form subsystems or a conceptual architecture. The approach was applied to a network protocol system and the result looked promising [Lung05b]. The main reason for the good result could be that the requirements for network protocols are well specified by the Internet community. The experiment was conducted after the system had been built for concept demonstration. It could provide some heuristics in system partitioning. However, the effect of this approach remains to be seen.

III. What Are the Gaps

Tremendous efforts are still under way in software architecture research. We identify some gaps based on our limited experiences and biased views. They are:

1. **Architecture-centric expert systems (ACES):** The knowledge or skills that a software architect should process are enormous. In practice, it is rare that people are skillful in both the problem and many of the solution spaces as well as the non-technical area. This is an unreasonable expectation. Various expert systems to support the tasks discussed in the previous section or other relevant areas can provide valuable information just in time. The expert systems should consist of concrete examples in addition to rules. In addition, those examples should compose real or realistic data obtained from similar systems, which can fill the gap between some modeling techniques and actual development. To support this step, we need to define requirements for data gathering for those modeling techniques, and more importantly, actually gather and characterize data for software systems or components.

2. **Generative frameworks based on well understood components or patterns**: They are needed to rapidly generate either a realistic prototype or a working system or sub-system to support effective design and efficient evaluation/comparison. Modeling techniques tend to deal with abstract data. Generative frameworks can provide more concrete or specific information. They can be filled in with more detailed information or design. After all, the devil is the details in practice. Patterns can be viewed as a result of domain analysis conducted by a number of experts. More fruitful results may be obtained by merging patterns and domain engineering disciplines. Techniques are more mature in generative approach or compositional approach to support some domain engineering and software reuse concepts. Generally speaking, patterns are robust and they are captured, partly, for reuse. The next step is to support effective development by reuse postulated by domain engineering.

3. **Architecture or program transformation in specific scopes or domains**: If certain architecture is known better, how effectively can the designer transform the existing system to that one? In an exercise, we spent several months reengineering a distributed and concurrent system mainly based on a well know design pattern, Half-Sync/Half-Async [Schmidt00]. In fact, in the post-mortem analysis, we identify that many changes are related to concurrency control, which mostly can be mechanically transformed. With the support of architecture transformation, it is easier to provide insights for architecture assessment and comparison.

4. **Ease of use and usefulness**: There is a dilemma in what we have been doing. On one hand, we need powerful techniques that allow us to capture detailed or realistic data to provide more accurate information. On the other hand, we want the techniques to be easy to understand and use. This is another gap in practice.

   Many methods and technologies have been developed but not adopted, including ours. One reason may be that we have focused too much on technology itself. We can learn from lessons presented in business management on technology adoption. For instance, Davis [1989] proposed that two particular factors, ease of use and usefulness, form a person's attitude toward adopting a technology. Other important factors reported in this field include relative advantage, compatibility, trialbility, visibility, results demonstrability, external pressure (voluntariness), demographics such as age or education [Moore91, Morris00]. Take telecommunications or embedded systems as an example, many architects had electrical engineering background and some of them may not have up-to-date software engineering trainings since. Asking them to adopt some advanced software technologies may be challenging if the technologies are not easy to use or provide useful information they need. Time-to-market also has a crucial role in the success of a system, which is directly related to ease of use and usefulness.

   Those reports may provide valuable hints. As we are trying to "push" technologies into the software community, we may need to frequently ask ourselves, "How should we "pull" from our customers (designers or other stakeholders) to find out more about what they need and what are really useful for them?"

References:

[Alexander64] C. Alexander, *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA, 1964.

[Arora95] V. Arora, K. Kalaichelvan, N. Goel, and R. Munikoti, "Measuring High-Level Design Complexity of Real-Time Object-Oriented Systems", *Proc. of Annual Oregon Workshop on Software Metrics*, pp. 91-94, 1995.

[Bot96] B. Sonia, C.-H. Lung, and M. Farrell, "A Stakeholder-Centric Architecture Analysis Approach", *Proc. of the 2nd Int'l Software Architecture Workshop,* pp. 152-154, 1996.

[Davis89] F. Davis, "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology", *Management of Information Science Quarterly*, vol. 13, no. 3, pp. 319-339, 1989.

[Gacek95] C. Gacek, A. Abd-Allah, B. Clark, B. Boehm, "On the Definition of Software System Architecture", *Proc. of the 1st Int'l Software Architecture Workshop*, April 1995.

[Hobbs05] C. Hobbs, "Architectures and tools for Sufficiently-Available Software", Internal Training Courses, Nortel Networks, 2005.

[Kazman94] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A Method for Analyzing the Properties Software Architectures", *Proc. of ICSE 16*, pp. 81-90, 1994.

[Kazman 96] R. Kazman, G. Abowd, L. Bass, P. Clement, "Scenario-Based Analysis of Software Architecture", *IEEE Software*, pp. 47-55, Nov 1996.

[Kazman98] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method", *Proc. of the 4th Int'l Conf. on Eng. of Complex Comp. Sys.*, pp. 68-78, Aug. 1998.

[Kostelijk05] T. Kostelijk, "Misleading Architecting Tradeoffs", *Computer*, pp. 20- 26, May 2005.

[Kruchten95] P. B. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, Nov. 1995, pp.42-50.

[Lung97] C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An Approach to Software Architecture Analysis for Revolution and Reusability," *Proc. of CASCON*, pp. 144-154, *1997*.

[Lung98a] C.-H. Lung, A. Jalnapurkar, and A. El-Rayess, "Performance-Oriented Software Architecture Analysis: an Experience Report", *Proc. of the 1st Int'l Workshop on Soft. Performance*, pp. 101-104. Oct 1998.

[Lung98b] C.-H. Lung, "Software Architecture Recovery and Restructuring through Clustering Technique", *Proc. of the 3rd Int'l Software Architecture Workshop (ISAW)*, Nov. 1998, pp.101-104.

[Lung00] C.-H. Lung and K. Kalaichelvan, "A Quantitative Approach to Software Architecture Sensitivity Analysis", *Int'l Journal of Software Eng and Knowledge Eng*, vol. 10, no. 1, pp. 97-114, Feb 2000.

[Lung02a] C.-H. Lung, A. Nandi, and M. Zaman, "Applications of Clustering to Early Software Life Cycle Phases", *Proc. of the Int'l Conf. on Software Eng Research and Practice (SERP)*, June, 2002, pp. 625-631.

[Lung02b] C.-H. Lung, "Agile Software Architecture Recovery through Existing Solutions and Design Patterns", *Proc. of 6th IASTED Int'l Conf. on Software Engineering and Applications (SEA)*, Boston, MA, Nov. 2002, pp. 539-545.

[Lung04] C.-H. Lung, M. Zaman, and A. Nandi, "Applying Clustering Techniques to Software Architecture Partitioning, Recovery and Restructuring", *Journal of Systems and Software*, vol. 73, no. 2, Oct 2004, pp. 227-244.

[Lung05a] C.-H. Lung and M. Zaman, "Application of Design Combinatorial Theory to Scenario-Based Software Architecture Analysis", to appear in the *Int'l Conf. on Software Eng and Knowledge Eng*, July 2005.

[Lung05b] C.-H. Lung, X. Xu, and M. Zaman, "Attribute Driven Software Decomposition", to appear in the *Int'l Conf. on Software Eng and Knowledge Eng*, July 2005.

[Moore91] G. C. Moore and I. Benhasat, "Development of an Instrument to Measure the Perceptions of Adopting an Information Technology Innovation", *Information Systems Research*, vol. 2, no. 3, pp. 192-222, 1991.

[Morris00] M. G. Moore and V. Venkatech, "Age Differences in Technology Adoption Decisions: Implications for a Changing Work Force", *Personnel Psychology*, vol. 53, no. 2, pp. 375-403, 2000.

[Schmidt00] D. Schmidt, et al., *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000.

[Smith90] C. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[STRAW01] From Software Requirements to Architectures Workshop (STRAW), 2002.

[STRAW02] From Software Requirements to Architectures Workshop (STRAW), 2002.

[Yu00] H. Yu and A. Vahdat, "Building Replicated Internet Services Using TACT: Tunable Availability/Consistency Tradeoffs." Proc. of the *2nd Int'l Workshop on Advanced issues of E-Commerce and Web-Based Information Systems*, June 2000.

[Zaman99] M. Zaman, C.-H. Lung, and A. Nandi, "Automated Software Architecture Partitioning and Visualization Using Arch", *Nortel Design Forum*, 1999.