Running Head: MODELING ARCHITECTURE CHARACTERISTICS

MetaTech Consulting, Inc.

White Paper

Appraisal of Techniques for Modeling Functional and Non-Functional Characteristics of Architectures for Software Intensive Systems

Jim Thomas

August 9, 2003

Abstract

This essay presents a treatment of recent research advances supporting modeling of architectures of software intensive systems to ascertain their functional and nonfunctional characteristics. The relationship between the architectural models and performance models is explored to support a discussion of the motivation driving further research on the subject. Generalized categories of modeling approaches are provided to provide a framework for discussions of the ongoing research. For each of the three categories identified, one or more efforts are summarized to exemplify the distinguishing characteristics, its strengths, and weaknesses. The essay concludes with an assertion supporting continued research in architectural modeling in general, and reliability prediction techniques in specific.

Appraisal of Techniques for Modeling Functional and Non-Functional Characteristics of Architectures for Software Intensive Systems

Building software intensive systems is a costly undertaking often involving multiple teams having disparate skills, objectives, and concerns. Models – abstraction representations of concrete *things* – are tools that facilitate effective communication between the involved parties and thereby increase the probability that a consistent vision is shared by all.

Models have additional value beyond this. As the complexity of the system increases, as the number of components and the interfaces between them increase combinatorially, it soon becomes impossible for its designers to maintain a single accurate and complete representation. Gokhale and Trivedi (2002, p. 1) assert "The size and complexity of computer systems has increased more rapidly in the past decade, than our ability to design, test, implement and maintain them." Models provide a means of reducing the apparent complexity of a problem through abstraction. Effective models have but a single purpose and make explicit only those characteristics necessary to communicate the necessary details – all other details are omitted purposefully. Multiple models are generally necessary to convey a complete representation of even the most simplistic system.

For many years, models representing flow control through software components together with the algorithms embodied in the processes have been employed to evaluate functional performance characteristics of software. Such modeling was performed as a testing strategy following the design phase of development. Pressman (2001) has illustrated the value of such rigorous modeling and testing as relative cost savings over the full life-cycle of a system (Figure

1). Later discussions within this paper represent the ongoing efforts to identify errors prior to the design phase through modeling of the architecture itself. Validation of functional (e.g. it does what it is supposed to do) and non-functional requirements (i.e. reliability, availability, security, etc.), prior to beginning the detailed design of the system is the objective.

Motivation for Modeling the Architecture

The practice of modeling software designs is as mature as software development itself. Through Computer Aided Software Engineering (CASE) tools software engineers are able to identify the flow control through software components and the interfaces each component exposes to the environment. Rigorous development processes often require software engineers to design the test plan in tandem with the software. The software development taking place in the context of a similarly rigorous process generally is done with two separate development teams: one team generates the software based on the software design specifications and the other generates the test code, based on the test plan together with the software design specifications.

Unlike the software design, the discipline of software architecting is relatively immature. Though there is increasing understanding of the value of rigorous architectural practices, there are few tools or techniques available to aid the architect in performing comparative analysis of architectural alternatives or for validating architectural decisions. The usefulness of modeling software architecture includes (a) validation of the correctness of the architecture, (b) facilitating tradeoff analysis of alternate architectures, (c) early identification of problems, and ultimately (d)

improving the probability of success of the software development activity. Each of these is discussed subsequently.

Validation of Architecture Correctness

The architect works closely with the client (e.g. the user) or the system to capture the objective characteristics of the system. The client's business objectives are articulated in requirements covering performance, security, reliability, scalability, availability, etc. The architect employs a number of models to communicate these objective requirements to both the client and the engineering team. Though examination of the models, it is possible to ensure that the architecture in fact addresses client's objectives – that the right problem is being solved.

Alternative Architecture Tradeoff Analysis

Very often, a problem has more than one solution. Each alternative solution has specific characteristics relative to cost, performance, extensibility, reliability, etc. Each of these characteristics represents concerns of the client. Through building models for a limited number of candidate architectures, it is possible to assess each to identify, with some confidence, how well each addresses the clients concerns. The architect can then work with the client to perform an architectural tradeoff, sensitive to the client's priorities, to select the best candidate architecture. Higher level architectural styles (Shaw & Garlan, 1996) provide manageable, recognizable constructs for performing much of this tradeoff analysis. For example, the client's business objectives may emphasize security above performance. The architect may then suggest a solution employing Common Gateway Interface (CGI) construct rather than a Java Applet (Fukuzawa & Saeki, 2002).

Early Identification of Problems

The preceding paragraphs describing the usefulness of architectural modeling has focused on the interface between the architect and the client. Modeling of architectures also has great value in agreements formed between the architect and the engineering team. Following initial selection of architectural constructs, the engineering team can begin to decompose and further refine each. Finer grain components such as architectural design patterns (Buschmann, Meunier, Rohnert, Summerlad, & Stal, 2001) provide meaningful level of resolution appropriate for these discussions. It is possible to perform comparative analysis to determine if the component's performance is consistent with the higher-level architectural model. Alternative architectural constructs, patterns, and styles can be considered to address problems at this stage prior to completing the detailed design.

Improve Probability of Success

Maier and Rechtin (2002) provided a thorough treatment of the utility of heuristics as they apply to the probability of success of architecting systems. These heuristics are themselves governing principles for architectural models. Many provide guidelines that the architect can use to evaluate his architectural models. For instance, after modeling an architecture, it becomes apparent of it is possible to achieve high cohesion and low coupling between the components – a foundation heuristic for successful software systems.

More discrete architectural models, such as those describe later in the present paper, make evident more detailed characteristics of software architecture. Activities comparable to the white-box analysis familiar to the software engineers provide more empirical methods of

evaluating architectures for probability of success (as measured by satisfaction of the clients concerns).

Classes of Models

Researchers have made substantial progress in the past several years in identifying meaningful modeling techniques for software architecture. Collaborative and independent work has resulted in a number of differing approaches including (a) simulation, (b) reliability prediction analysis, and (c) scenario-based analysis. A description of each class of approach for architecture modeling is provided in the remaining portion of this paper. One or more examples of each class is discussed to highlight its characteristics. Also, the strengths and weaknesses of each class will be addressed independently for each class.

Simulation

This class of models strives to represent the software architecture in a language that is compatible with existing simulation tools. Fukuzawa and Saeki (2002) have successfully demonstrated simulation of architectures by using formal architecture definition language (ALD) representations of architectures in the context of Coloured Petri Nets (CPNs) for the purpose of evaluating alternatives. CPNs employ tokens that are used to persist quality attribute as they transverse the architectural components. As a transition occurs, the quality attributes are recalculated and applied to the token. Functional (i.e. behavioral) and non-functional (reliability, security, and efficiency) characteristics can be determined for candidate architectures.

Efforts at the Laboratory for Intelligent Processing located at the University of Texas at Austin (Barber, Holt, & Baker, 2002) have resulted in an architecture evaluation tool called Arcade. This tool generates from the ADL representing an architecture tokens appropriate for use by the commercial SimPack simulation environment. The Arcade tool automates the otherwise manual processes of translating the ALD to the token suitable for the simulation environment (Figure 2). Additionally, the tool automates the collection of simulation results and provides them in an intuitive representation.

Strengths. Tools enable a given unit of work to be completed with less effort or in less time. Simulation employs tools to the greatest extent possible. Research is advancing the capabilities to tool to produce accurate and complete ADL suitable for a preexisting selection of simulation environments. As these tools mature, it will become easier to quickly and reliably generate ALD for architectural styles and patterns and to test their reliability.

Weaknesses. Based on the research performed, this author asserts the state-of-the-art in architectural simulation addresses only the bridge between the architect and the engineer: it does not address the necessary communication between the architect and the client. Though both of the cited examples of architecture modeling through simulation do rely heavily on scenarios derived from the clients business needs, they can be employed only after substantial design (e.g. engineering) effort has been completed.

Reliability Prediction Analysis

As stated previously, large scale development of software intensive systems is generally accomplished through multiple teams of developers. Each team will endeavor to build one or

more components that must interact with the components built by other teams. It is not until integration testing, the final phase of development, that it becomes possible to test how the system works as a whole. Reliability prediction analysis modeling techniques concern themselves with representing system capabilities (e.g. reliability, scalability, etc.).

Through direct measurement or estimation, the models capture metrics for each component such as the length of execution time, the probability of and frequency of execution, and the probability of inter-component transition. Finding for individual components are then aggregated to create estimations of the system as a whole. Gokhale and Trivedi (2002) provide a discussion of two differing methods of generating these models: a composite method that generates more accurate results, and a hierarchical method that produces approximate solutions but that is easier to implement. The determination as to which method is most appropriate is left to the practitioner as a subjective exercise.

Strengths. This class of architecture modeling techniques excels at identifying performance and reliability bottlenecks. It incorporates algorithmic techniques proven in modeling non-functional characteristics of systems and, while it recognizes the tendency of individual components to fail, it also factors the impact of single failures to the reliability of the system as a whole. Furthermore, this modeling technique has application throughout the development life-cycle.

Weaknesses. Reliability prediction analysis modeling of software architectures is more accurate when employed in the later stages of the software development life-cycle. Though the title of the work published on the subject by Gokhale and Trivedi (2002) emphasizes its

predictive capability, the content emphasizes its strength when applied in the development and operational phases of the life-cycle.

Scenario-based Analysis

A third approach to evaluating architectures utilizes usage scenarios as the analytical vehicle. The Performance Assessment of Software Architectures (PASA) technique explained by Williams and Smith (2002) provides a stepwise approach that identifies and documents as scenarios the critical workloads of the objective system. These scenarios are then used to capture the performance criteria in precise, quantitative, and measurable terms.

Together, the scenarios and their respective performance criteria as applied to the proposed architecture. The analysis reveals areas where the architecture does not support the criteria. Architectural alternative are proposed and evaluated in a stepwise, iterative process. Upon reconciling the architecture with the performance criteria, additional economic analysis is performed to ascertain the costs and benefits of the proposed solution.

Strengths. This approach to architectural analysis places the greatest weight on the client/user view of system performance and functionality. The client/user is involved throughout the PASA process to minimize dilution of her objectives. Additionally, this approach leverages known characteristics of architectural styles (Shaw & Garlan, 1996), patterns (Buschmann, Meunier, Rohnert, Summerlad, & Stal, 2001), and anti-patterns as described by Brown, et al. (1998) and cited by Williams and Smith (2002). Anti-patterns, patterns that have proven to result in negative performance characteristics, are valuable tools in the architect's repository that allow him to learn from other's mistakes and not just from their successes.

Weaknesses. It is worthwhile to make explicit that the PASA process is principally concerned with interactions between the user/client and the architect. While this interaction is critical, it is not sufficient to fully evaluate the architecture. Subsequent detailed analysis of the architecture from an engineering perspective would ensure the feasibility of proposed architecture.

Conclusions

As the discipline of software architecting continues to mature, researchers contribute to its value to the software development industry. Decades of research have resulted in effective tools that aide software engineers in measuring the effectiveness of their designs earlier in the life-cycle and thereby realize cost savings to the project. Only recently has this philosophy been extended to earlier still in the life-cycle – the architecture definition phase. As architecture modeling tools and techniques become more robust and capable the software development industry can expect even more cost savings, more reliable systems, and reduced development time. This essay has discussed three classes of modeling techniques that aim to determine the functional and non-functional (e.g. performance, reliability, scalability) characteristics of a given architecture. It is the opinion of this author that continued research on reliability prediction analysis modeling will likely result in significant greater utility to architecture practitioners than the alternative classes discussed in the present paper.

References

Barber, K. S., Holt, J., & Baker, G. (2002). Performance Evaluation of Domain Reference Architectures. *Proceedings of the 14th international conference on software engineering and knowledge engineering*, pp. 225 – 232. New York: ACM Press.

Brown, W.J., Malveau, R.C., McCormick, H.W., III, & Mowbray, T.J. (1998). *Antipatterns: Refactoring software, architecture, and projects in crisis*. New York: John Wiley and Sons, Inc.

Buschmann, F., Meunier, R., Rohnert, H., Summerlad, P., & Stal, M. (2001). *Pattern-oriented software architecture: A system of patterns*. New York: John Wiley & Sons Ltd.

Fukuzawa, K. & Saeki, M. (2002). Eavaluating software architecture by coloured Petri nets. *Proceedings of the 14th international conference on software engineering and knowledge engineering*, pp. 225 – 232. New York: ACM Press.

Gokhale, S. S., & Trivedi, K. S. (2002). Reliability prediction and sensitivity analysis based on software architecture. *Proceedings of the 13th international symposium on software reliability engineering*, pp. 1 – 12. Washington, DC: IEEE Computer Society.

Maier, W.M., & Rechtin, E. (2002). *The art of systems architecting (2nd ed.)*. Boca Raton, FL: CRC Press LLC.

Pressman, R. S. (2001). *Software engineering: A practitioner's approach (5th ed.)*. New York: McGraw-Hill.

Shaw, M., & Garlan, D. (1996). *Software architecture: Perspectives on an emerging discipline*. Upper-Saddle River, NJ: Prentice-Hall, Inc.

Williams, L.G., & Smith, C.U., (2002). PASA(sm): A method for performance Assessment of software architectures. *Proceedings of the third international workshop on Software and performance*, pp. 179 – 189. New York: ACM Press.

Figure 1

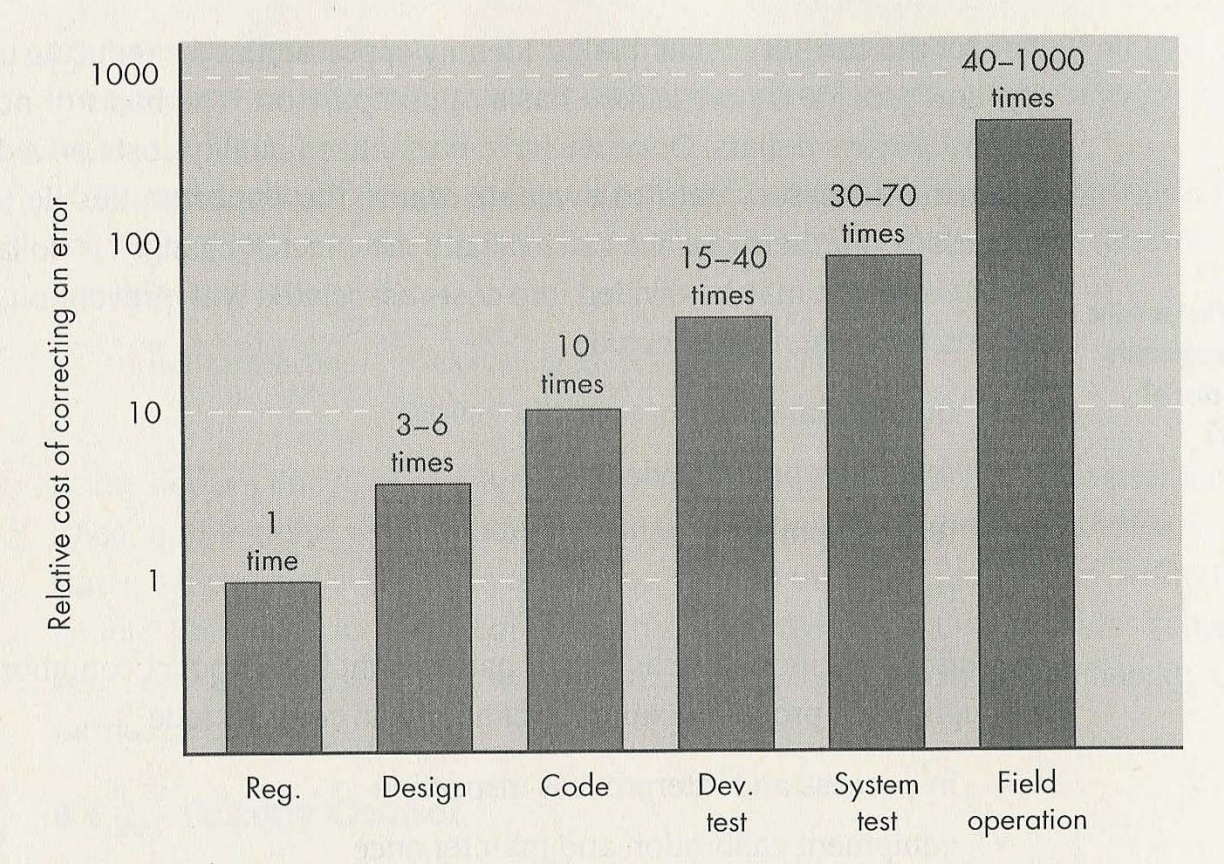

Figure 1. Relative cost of correcting an error (Pressman, 2001, p. 198)

Figure 2

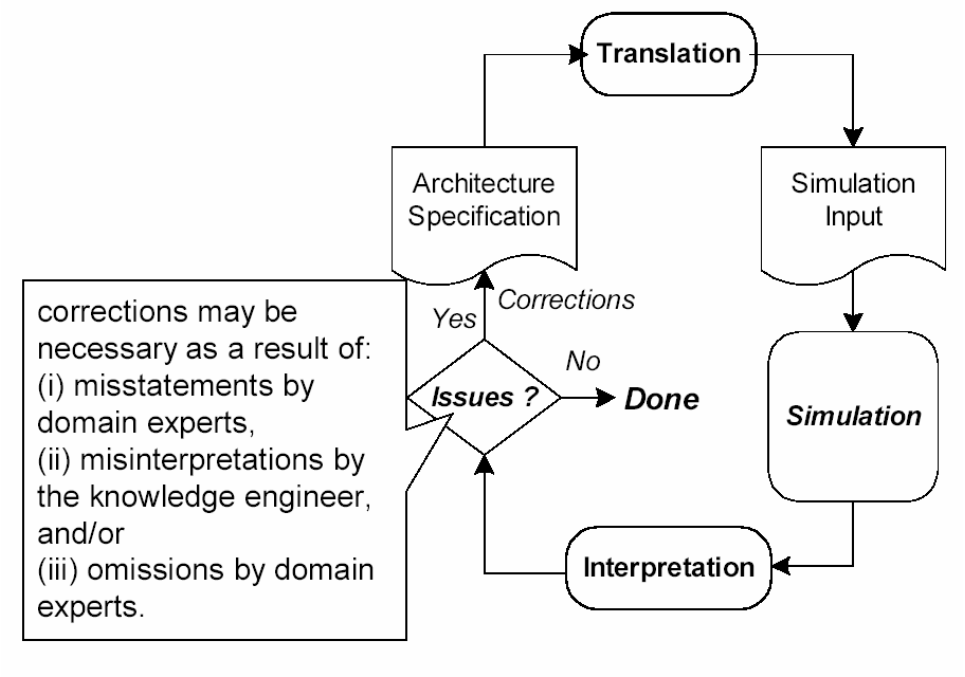

Figure 2. Iterative architecture evaluation process (Barber, Holt, & Baker, 2002, p. 226)