Software Architecture: Implications for Computer Science Research

C. Williams
IBM T. J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598, USA
(914)-945-1105, Fax: (914)-945-4017
clayw@us.ibm.com

Abstract

Software architecture has the potential to significantly shift software development from dealing with fine grained (program level) constructs to a higher level emphasis. This position paper discusses some of the major computer science questions that require investigation to facilitate this shift. It focuses on three particular areas: semantics and languages, formal verification, and measurement and metrics. These areas are examined as part of the scientific basis on which progress in software architecture relies. In each area, high level challenges are identified and potential research directions are discussed.

Keywords

Software architecture, granularity, semantics, languages, metrics, measurement, formal verification

1 INTRODUCTION

In (Shaw and Garlan, 1996), the authors discuss a two-phase model for the emergence of a professional engineering discipline. The first phase is the transition from a craft, which relies on virtuoso effort and haphazard methods, to routine production. As a result of routine production, the second phase occurs, which is the transformation of routine production into engineering via the development and incorporation of a supporting science. Software development has completed the first phase of this process, and the second is underway. This

paper examines some of the questions that will be posed to computer science as a result of the emergence of software architecture as a sub-discipline of software engineering. All of these questions have their roots in the issue of trying to raise the granularity used when building software systems.

The Problem of Granularity

The need to increase the level of abstraction used when developing software has long been acknowledged. However, efforts to raise the level of granularity at which software development occurs have met with mixed results at best. In spite of these efforts, industrial software construction deals effectively with abstraction only at the program level.

In the paper below, I explore three areas of computer science that could help software architecture raise the level of granularity involved in developing software. These are semantics and languages, formal verification, and measurement and metrics.

2 SEMANTICS AND LANGUAGES

The study of semantic structures has traditionally focused on mathematical and programming constructs at a very fine level of granularity. Program semantics are typically studied at the level of repetition structures (Dijkstra, 1990), categories and data types (Mitchell, 1996), abstract machines (Plotkin, 1982), continuous functions and least fixed points (Scott, 1982), and other fine grained structures. While the gains from these studies have been impressive and significant, understanding semantics at a higher level of granularity is required for progress in software architecture. Thus, an important part of the science that must emerge to support software engineering is the semantics of complex architectural descriptions.

Developing semantic theories of more complex language constructs is a challenging and difficult enterprise. For example, questions concerning the semantics of method specialization and inheritance remain open (Mitchell, 1990). The development of software architectures using architectural definition languages (ADLs) will provide a set of difficult semantic issues at a significantly higher level of granularity and complexity than has been effectively dealt with thus far. These issues will foster further research in the theory of semantics and languages.

Example: Design Patterns

An example of an area where semantic progress will be needed is the field of design patterns. Design patterns are abstract representations of commonly occurring solutions for software architectural issues. Catalogs of design patterns are emerging; (Gamma et. al., 1995) is a source for design patterns dealing with

object-oriented architectures. (Shaw and Garlan, 1996) argue that no single semantic framework will be capable of providing the variety of analytic capabilities that might be desired for architectural descriptions in an ADL. Thus, at least some of the semantics must be specified as needed. This raises two issues.

First, as commonly used design patterns are discovered and documented, basic semantics should be specified for the pattern and supported in ADLs that typically use the pattern. For instance, the Singleton pattern (Gamma, et. al., 1995) provides a mechanism to guarantee that a class has exactly one instance, and provides an easily available access point for this instance. Thus, basic semantic invariants about objects that are Singletons can be derived. One such invariant is that an object that is an instance of a Singleton must be the only instance. This can
be represented using many formal semantic notations. Examining design patterns to identify fundamental properties, and the design and development of basic semantic representations for these properties is an important task in ADL research.

The second issue involves semantic properties that are not fundamental to a component, but might be needed to perform some desired analysis using an ADL. Just as ADLs should make it simple to represent patterns as first-class abstractions (Shaw and Garlan, 1996), specialized semantics should be represented in a manner that provides a unified view of the general and specialized semantics of the component in question. For example, suppose that a data repository is going to be represented in an architecture as a Singleton pattern. If the component represented by the pattern has complex relationships in the architecture (such as providing atomic operations on data to several other components), the details of these relationships need to be specified in such a way that analysis can use them as well as the implicit, fundamental semantic notions already developed for the pattern. The development of powerful and efficient representations for specialized semantic information is a second task that ADL research must address.

3 VERIFICATION

Closely related to the area of semantics for reasoning about architectural descriptions is the notion of verification. Verification techniques exist for sequential, concurrent, and distributed programs (Apt and Olderog, 1991). These techniques rely on program level methods such as operational semantics and syntax-directed assertional proof systems. If software architectures are expressed in an ADL with well-defined semantic properties, the possibility of verifying certain properties of the architecture arises. Two issues arise concerning the

development of mathematically rigorous methods for performing these verifications.

The first concerns the goals of verification analysis. What are the properties of interest for a given architecture? There are several candidates to consider, and in some cases they are architecture dependent. For example, pipe and filter architectures might raise questions concerning deadlocks and fairness. Certain database architectures raise the issue of commutativity of operations and of atomicity. Experience in developing and analyzing architectures, as well as the structure of the semantics available, should be used to determine the goals of verification analysis for software architectures.

Given the goals of the verification analysis and the semantic structures of the ADLs that are used to represent software architectures, new techniques for verification will need to be developed. The development of concurrent programs required the addition or extension of various capabilities (such as compositionality and determinism) to sequential verification techniques. Similarly, higher level semantics of ADLs will require the development of novel proof methods for verifying the desired properties of software architectures.

4 MEASUREMENT AND METRICS

The efforts of software engineers to efficiently build economical, reliable, and maintainable systems has lead to a plethora of models and metrics concerning issues such as productivity, complexity, cost estimation, and reliability (good surveys include (DeMarco, 1982) and (Conte, et. al., 1986)). Critiques of and proposals for metrics research also abound, such as (Kitchenham, et. al., 1995). If attempts at developing a scientific basis for software architecture are successful, measurement will be affected profoundly. Two specific challenges are discussed below.

The first challenge will be to change what is being measured. Despite the best efforts of many researchers, the fundamental models and metrics used in the industry today still rely on lines of code (LOC) and other program level constructs as the basis for measurement and prediction. The development of software architecture (and particularly component based architectures) will provide an unprecedented opportunity for fundamentally changing the field of software measurement. Models and measurements for software construction can be refocused around components, connectors, patterns, and their properties.

Beyond redefining what is being measured, software architecture might also have an impact on deeper, more fundamental questions concerning software. An ongoing question that arises as a result of measuring properties of software is the validity of treating software as a physical entity. Consider the field of software reliability (Musa, Iannino, and Okumoto, 1987). Is developing reliability models for software systems using assumptions that are typically valid for physical

systems (such as hardware) the best way to proceed? Or does software, as a conceptual rather than physical product, have different properties on which we should focus? Software architecture may be important in considering questions like this one, because it raises the granularity considerably as we consider the conceptual entities that are software components. Fundamental properties exist for physical components (resistivity, modulus, melting point, etc.), and can be used for reasoning about a physical system composed of such components. A goal of software architecture should be to explore software components, connectors, and patterns for fundamental properties that will provide insight into a system composed of the components. Perhaps such properties will be analogous to properties in the world of physical systems, but we should be sensitive to the fact that they might be markedly different.

5 CONCLUSION

In this paper, I examined three potential areas where the development of software architecture might spur new research in computer science to provide an avenue for developing software systems at a higher level of granularity. These were semantics and language theory, verification, and measurement and metrics. Several potential topics of investigation were proposed.

In the area of semantics and language design, we need to explore design patterns and develop sets of semantic constructs for ADLs that are sufficient to capture fundamental properties of the patterns. We also need capabilities for capturing specialized (architecture dependent) semantics and treating them in a unified manner with the fundamental constructs.

In verification, we must determine the goals that verification has when examining software architectures. Techniques that support these goals will need to be explored, and new techniques for verifying software architectures will need to be developed.

Finally, software architecture provides new opportunities in the area of models and metrics. First, changing the practice of measuring and building models based on LOC can be a goal of software architecture. The deeper question of the fundamental nature and properties of software components may also be affected by developments in software architecture. The impact of changes in measurement practice offered by software architecture is directly applicable to industrial issues faced today.

In exploring these three areas, the breadth and scope of the challenges facing software architecture are demonstrated. Direct, near term enhancements to industrial practices are one result of architectural research, yet at the same time, software architecture raises profound theoretical questions. Just as high level programming languages based on sound theoretical concepts helped make routine software production possible, developing sound theories of higher level

architectural constructs will be an important part of the effort to create an engineering discipline for software construction.

6 REFERENCES

Apt, K.R. and Olderog, E. (1991) *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York.

Conte, S.D., Dunsmore, H.E., and Shen, V.Y. (1986) *Software Engineering Metrics and Models*. Benjamin/Cummings, Menlo Park, CA.

DeMarco, T. (1982) *Controlling Software Projects*. Prentice-Hall, Englewood Cliffs, NJ.

Dijkstra, E.W. and Scholten, C.S. (1990) *Predicate Calculus and Program Semantics*. Springer-Verlag, New York.

Kitchenham, B., Pfleeger, S.L., and Fenton, N. (1995) Towards a framework for software measurement validation. *IEEE Trans. on Soft. Eng.*, **21**, 929–44.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Mitchell, J.C. (1990) Toward a Typed Foundation for Method Specialization and Inheritance, in *Principles of Programming Languages* (ed. P. Hudak), 17th ACM Symposium: Principles of Programming Languages, San Francisco, CA.

Mitchell, J.C. (1996) *Foundations for Programming Languages*. MIT Press, Cambridge, MA.

Musa, J.D., Iannino, A., and Okumoto, K. (1987) *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, New York.

Plotkin, G.D., (1982) An Operational Semantics for CSP, in *Formal Description of Programming Concepts II* (ed. D. Bjorner), Proc. of IFIP Working Conference, Garmisch-partenkirschen, Germany.

Scott, D.S. (1982) Domains for Denotational Semantics, in *Springer Verlag Lecture Notes in Computer Science vol. 140*, Springer-Verlag, New York.

Shaw, M. and Garlan, D. (1996) *Software Architectures: Perspectives on an Emerging Discipline*. Prentice-Hall, Upper Saddle River, NJ.

7 BIOGRAPHY

Clay Williams is an Advisory Software Engineer in the Center for Software Engineering at IBM T.J. Watson Research Center. He received his Ph.D. in

computer science from Texas A&M University in 1994. He is a member of the ACM and the IEEE.