On the Definition of Software System Architecture

Cristina Gacek
Ahmed Abd-Allah
Bradford Clark
Barry Boehm

Center for Software Engineering
University of Southern California
Los Angeles, CA, 90089-0781

ICSE 17 Software Architecture Workshop
April, 1995

Abstract

Although several definitions of "software architecture" have been presented, none of them to date enable a reviewer confronted with a complex of diagrams and symbols to determine whether it is an architecture for a system or not. We present a definition of "software system architecture" which provides a set of criteria for making this determination. It is based on making the architectural rationale a first-class citizen in the definition, and on requiring the rationale to ensure that the architecture's components, connections, and constraints define a system that will satisfy a set of defined stakeholder needs for the system.

1. Introduction

Intuitively people apply the general term architecture to the usage aspects of the houses and other buildings they deal with, in terms of the nature of the physical structures and the physical arrangement of the structures in relation to each other. Whereas this is the general individual perspective of architecture, there is another very important one, that of the building architect. This perspective involves the building architect's need to produce a building which simultaneously satisfies a number of stakeholder needs involving shelter, light, heat, accessibility, safety, aesthetics, maintainability, communication, etc. [Alexander 1964]. Other important perspectives are those of building contractors, building inspectors, safety engineers, or urban planners, each of whose missions rely on architectural information.

Software architecture has been defined in various ways. It has been defined as a structure composed of components, and rules characterizing the interaction of these components [Jones 1993]. It has been defined as components, connections, constraints and rationale [Boehm 1993]. It has also been defined as elements, form, and rationale [Perry and Wolf 1992]; and as components, connectors, and configurations [Garlan and Shaw 1993]. These definitions appear to focus on what can be

seen " from the street," or architectural representation, but it is not clear that they fully address the full range of evaluation issues associated with a software architecture.

As a major example, a fundamental practical question involving software architectures is: Does a given system under construction (e.g. the FAA Advanced Automation System, the Denver Airport baggage handling system) have a software architecture or not? Is any collection of diagrams and symbols an architecture? Is it sufficient (or necessary) for the collection to be syntactically consistent?

We claim that the candidate definitions above do not provide evaluators with enough information to satisfactorily answer these questions. We further claim that the following definition of a " software system architecture" does provide the requisite information.

A software system architecture comprises:

- A collection of software and system components, connections, and constraints.
- A collection of system stakeholders' need statements.
- A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.

This paper proceeds to elaborate this definition and its context.

In section 2 we discuss existing definitions of software architecture. In section 3 we elaborate the stakeholder's needs aspect of our definition of software system architecture. In section 4 we discuss the critical role of the software systems architecture in the software process. In section 5 we discuss some implications of the nature of software architectural representations, particularly in supporting the views necessary to represent a software system architecture. In section 6 we present our conclusions.

2. Previous definitions

One of the earliest definitions of software architectures, by Perry and Wolf [Perry and Wolf 1992], has remained one of the most insightful. After examining the architectures of other disciplines (e.g. hardware, networks, and buildings), Perry and Wolf describe a software architecture as " a set of architectural (or if you will, design) elements that have a particular form." The elements are divided into three classes: processing elements, data elements, and connecting elements. Whereas the processing and data elements have been studied extensively in the past (e.g. functions and objects), it is the connecting elements that especially distinguish one architecture (or style) from another. The form of the architecture is given by enumerating the properties of the different elements and the relationships between the elements. Another essential part of the architecture is its rationale which includes quality attribute aspects among other things.

Perry and Wolf also define an architectural style as " that which abstracts elements and formal aspects from various specific architectures." Thus, an architectural style consists of a set of shared assumptions and constraints across a set of architectures. An architectural style is not an architecture; a point of confusion in a number of presentations. The utility of a particular style comes from its addressing important classes of design decisions up front, isolating and highlighting certain aspects. A style or a specific architecture can be viewed in three diferent ways based on the elements: a processing view, a data view, and a connections view. All three views are necessary for the understanding of the style or architecture.

Another significant definition of software architecture has been advanced by Garlan and Shaw [Garlan and Shaw 1993]. It is more restrictive than the definition of Perry and Wolf. Garlan and Shaw proposed that a software architecture for a specific system be captured as " a collection of computational components - or simply *components* - together with a description of the interactions between these components - the *connectors*." Based on this definition, the authors used the term architectural style to denote a family of systems (architectures) that share a common vocabulary of components and connectors, and which meet a set of constraints for that style. The constraints can be on a variety of things, including on the topology of connectors and components, or on the execution semantics.

To instantiate the concepts of architecture and style, Garlan and Shaw presented a valuable partial taxonomy of known architectural styles (see Table 1).

Pipes and Filters	Layered	Distributed
Object-Oriented	Repositories	Main program/subroutines
Event-Based	Rule-Based	State transition based
Domain-Specific	Process Control (Feedback)	Heterogeneous

Table 1: Some known architectural styles [Garlan and Shaw 1993]

For each style, they asked questions designed to bring out its unique characteristics such as " What is the structural pattern of components and connectors?" and " What are some common examples of its use?" However, no particularly formal classification of styles was presented.

It is important to note that quality attributes are not directly addressed by the [Garlan and Shaw 1993] view of software architecture—in fact, the rationale found in Perry and Wolf's definition is entirely missing. The focus is also primarily on the structure of the software, with little characterization of to the overall environment in which the software operates.

Another source of software architecture definitions is the ARPA Domain Specific Software Architecture (DSSA) program. While a general definition of what is an architecture has been laid down, the exact form varies from project to project within the DSSA program. In general, a software architecture is defined as

> " ...An abstract system specification consisting primarily of functional components described in terms of their behaviors and inter-

> faces and component-component interconnections. The interconnections provide means by which components interact. Architectures are usually associated with a rationale that documents and justifies constraints on component and interconnections or explains assumptions about the technologies which will be available for implementing applications consistent with the architecture."[Hayes-Roth 1994]

A *domain-specific* software architecture, on the other hand, consists of a software architecture as well as a domain model and a set of standardized requirements. The DSSA definitions seem to combine aspects of Garlan/Shaw and Perry/Wolf, plus a little extra in the case of a domain-specific architecture. Garlan and Shaw's definition of architectural style has been more or less adopted within the DSSA program [Tracz 1994].

In spite of their common definition, the different DSSA projects have yielded architectures whose representations focus on different aspects of systems. For example, the TRW/Stanford project has produced architectures expressed in RAPIDE, an " event-based concurrent, object-oriented language specifically designed for prototyping system architectures."[Luckham et al. 1994] The Honeywell/Maryland project, on the other hand, has focused on and produced architectures for intelligent guidance, navigation, and control (GNC) applications. Consequently, their architectures are expressed in ControlH and MetaH, two languages that focus on GNC and scheduling issues respectively [Binns et al.]. A third project from Teknowledge/Stanford has generated architectures for autonomous vehicles expressed in ArTek, yet another distinct language [Terry et al. 1993].

Some further definitional considerations resulted from a consensus reached in the Process-sensitive SEE Architecture workshop held in 1992. There, Penedo and Riddle included some architectural aspects which we believe the target definition should include [Penedo and Riddle 1993]. They concluded that an architecture should be viewed and described from different perspectives, and should identify its components, their static inter-relationship, their dynamic interactions, properties and characteristics, and constraints on these items.

Although this definition is considerably richer than the ones previously discussed, it is too flexible on allowing for any kind of interpretation for " properties and characteristics". It also lacks specificity on determining what actually are the different perspectives required for viewing the architecture.

3. Elaboration of "Software System Architecture" Definition

This section elaborates on the " stakeholder needs"portion of the definition of " software system architecture"presented in Section 1. As with physical systems such as buildings, different stakeholders in the software lifecycle take different viewpoints when expressing their concerns about a software system. These viewpoints reflect the stakeholders' differing needs with respect to the software system architecture. Given that stakeholders needs will vary from system to system, the

software system architecture's emphasis will also vary from system to system.

3.1 Stakeholders and their Architecture Needs

Software architecture has a different meaning and use for different stakeholders; see Table 2. The *customer* may expect at the architecting stage an estimate

Stakeholder	Concern
Customer	• Schedule and budget estimation • Feasibility and risk assessment • Requirements traceability • Progress tracking
User	• Consistency with requirements and usage scenarios • Future requirement growth accommodation • Performance, reliability, interoperability, etc.
Architect and System Engineer	• Requirements traceability • Support of tradeoff analyses • Completeness, consistency of architecture
Developer	• Suffi cient detail for design • Reference for selecting / assembling components • Maintain interoperability with existing systems
Maintainer	• Guidance on software modifi cation • Guidance on architecture evolution • Maintain interoperability with existing systems

Table 2: Stakeholder Concerns

of certain factors once the software structure has been defi ned. For example, the customer is likely to be concerned with getting fi rst-order estimates of the cost, reliability, and maintainability of the software based on its high-level structure. This implies that the architecture should be strongly coupled with the requirements, indicating if it can meet them. *Users* need software architectures in order to be able to clarify and negotiate their requirements for the software being developed, especially worrying about future extensions to the product. The user will be interested at the architecting stage in the impact of the software structure on performance, usability, and compliance with other system attribute requirements. As with architectures of buildings, users also need to relate the architecture to their usage scenarios.

Architects and *Systems Engineers* are concerned with translating requirements into high-level design. Therefore, they may use a software architecture for clarifying and negotiating the requirements of the system. *Developers* are concerned with getting an architectural specifi cation that is sufi cient in detail to satisfy the customer's requirements but not so constraining as to preclude equivalent but different approaches or technologies in the implementation. Developers then use the architecture as a reference for developing and assembling system compo-

nents, and also use it to provide a compatibility check for reusing pre-existing components. *Interfacers* use the software architecture as a basis for understanding (and negotiating about) the product in order to keep it interoperable with existing systems.

The *maintainer* will be concerned with how easy it will be to extend or modify the software, given its high-level structure. Software Architecture provides maintainers with a core structure to the software that should not be violated. It is the most important aspect of the software which maintainers ought to maintain with as few changes as possible at the architectural level, trying to restrict their changes purely to the component level. In the inevitable case of product extension, maintainers must attempt to extend the architecture in logical, reasonable ways. In all cases, we would like as precise a representation as possible to express the desired information. From the developer's view (or architect's view), this stage of the lifecycle is where a formal, high-level description of the software structure is given, both static (topological) and dynamic (behavioral). It is possible that other types of descriptions may be given as well.

4. Role of the Software System Architecture in the Lifecycle Process

Using our defi nition, the Software System Architecture can serve as the key milestone in the entire software life-cycle process. Until one has an architecture whose rationale ensures that it will satisfy the needs of the system's stakeholders, it is very risky to proceed into full-scale system development and evolution. Thus, the achievement of a Software System Architecture as defi ned here can and should be used as the precondition for transitioning from an uncommitted requirements / architecture exploration stage into a full-scale development and evolution stage based on a solid set of requirements / architecture commitments.

Before this milestone is reached the most effective process is generally a risk-driven spiral process particularly its recent extension into the stakeholder win-win spiral process [Boehm and Bose 1994]. As shown in Table 3, several (not necessarily three) spiral cycles are used to converge on a compatible set of objectives, constraints, and alternatives for the system's life-cycle concept of operation, requirements, architecture, and plans.. During this spiral process, these artifacts are selected and grown in detail as risks are identifi ed and resolved, and interactions among the artifacts are explored. Once such a Software System Architecture and its associated artifacts are in place, the project can use a waterfall, spiral evolutionary, or other selected process to pursue the post-architecture full scale development and evolution process

The Software System Architecture's support of the needs of system engineers, customers, developers, users, and maintainers, also implies that it is involved in all phases of the software and system life-cycle; see Figure 1.

For example, design, code and unit test involve elaboration of the details deferred by the risk-driven software system architecture specifi cation. Also, the software system architecture provides a strong framework for software system

Cycle 1	Cycle 2	Cycle 3
Determination of top-level concept of operations	Determination of detailed concept of operations	Determination of IOC requirements, growth vector
System scope/ boundaries/ interfaces	Top-level HW, SW, human requirements	Choice of life-cycle architecture
Small number of candidate architectures	Provisional choice of top-level information architecture	Some components of above TBD (low-risk and/or deferrable)
Top-level analysis supporting win-win satisfaction	More detailed analysis supporting win-win satisfaction	Thorough analysis supporting win-win satisfaction

Table 3: Spiral Model Task Decisions

integration. This reduces risk and test time. Software architectures work out in advance the major dependencies between system components. This provides a consistency of form, and partitions the complexity. The architecture defines the interfaces between components, and furthermore provides a basis for software maintenance in order to prevent architectural erosion and drift [Perry and Wolf 1992]. Enough violations of the product's original architecture eventually makes maintenance difficult.

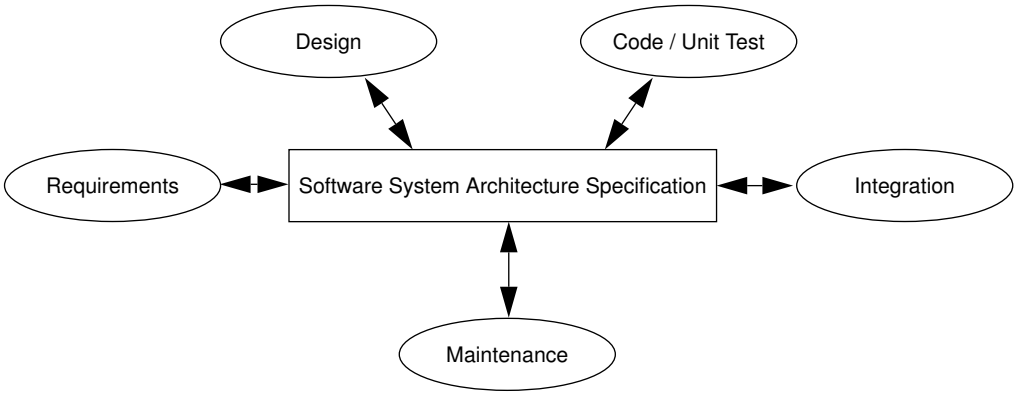

Figure 1. Lifecycle Involvement of Software System Architecture Specification

Whenever existing software has to be modified, it must be understood first before non-destructive changes can be made. Currently, software understanding consumes roughly 47% of the maintenance effort. The overall framework and its rationale carried by the software architecture will serve to reduce this cost significantly.

A software architecture should identify properties of components such as hardware platform, implementation language, communication mechanism, cost, and performance. It should be analyzable for quality, either by means of a formal

model or some other kind of representation or technique. Formal models also provide the basis for ways to verify an architecture, simulate its execution, enhancing or facilitating the analysis of factors such as portability, scalability, performance, robustness, and reusability.

Software architectures also facilitate the reengineering of legacy systems and the identifi cation of components for reuse. The power of reuse is greater the earlier it is done in the lifecycle. Since the architecting stage acts as a bridge between requirements analysis and software design, it is clear that reuse at the architectural level provides a tremendous amount of leverage for systems with similar requirements.

5. Representing Software Architectures

Our defi nition of " software system architecture"also implies that architectural representation schemes need to represent and to support reasoning about an architecture's ability to support stakeholder needs. The architecture should be comprised of alternate views, including at least a behavioral/operational view, a static topological view, and a dataflow view. It is important to have formal architectural notation(s) that are capable of capturing not only these views, but also *other views that are concerned with other stakeholder needs*. The notation also needs to include attributes which support reasoning about the cost, performance, reliability, portability, completeness, consistency and other stakeholder-critical properties of the system represented by the architecture.

5.1 Notations

Formal methods and notations for software design have been shown to aid the development of quality software. It is advantageous to use these techniques as early in the life cycle as possible: in software architectures. Most work in this area has focused on using formal methods to describe only the topological structure of the software. For example, the Unix operating system can be described as a series of concentric *layers* centered around the kernel, with each layer providing a different virtual machine to the layer above. Another example is found in many embedded controllers where the software is structured as a *single main component* which receives inputs from (or is interrupted by) sensors, and which reacts by sending outputs to actuators. In both examples we have only described, at a very high level, the static topological structure of the software, ignoring other vital issues.

As systems grow in size and complexity, it becomes increasingly important at the architecting stage to specify more than the topological structure of the product. To understand what else to specify, we must turn to the stakeholders' concerns in Table 1. The different expectations of the various stakeholders make it clear that a description of a software architecture must incorporate different, multiple " viewpoints". A recent survey of some known architectural description notations show a lack of support for some of these viewpoints; see Table 4 [Gacek et al. 1994]. Traditional defi nitions of software architecture have focused primarily on the developer's viewpoint. This ignores other issues vital to other aspects of the lifecycle

	MetaH	ControlH	DICAM	WRIGHT	UNICON	LEAP	Rapide	UNAS(Z)
Static structure (source topology)								
Dynamic structure (behavior)								
Dataflow								
Domain-specifi c info								
Domain-independence								
Implementation-dependent info								
Support formal analysis								
Executable								
Support reliability analysis								
Cost								
Formalize nonfunctional info								

Table 4: ADL Support for Stakeholder Views

such as the complexity of the algorithms, execution sequence, user interaction, and resource constraints. Just as a program can be viewed from different perspectives, the same with a software architecture. While the high-level representation of the software structure is the cornerstone of the architecture, it is not suffi cient towards guiding the quality development of systems, especially large systems. At the architecting stage, it is necessary for large systems to compare different software structures, weighing them against the different concerns of all the stakeholders

Given that multiple viewpoints must be supported in a full architectural description and that most large systems are composites of different topological structures, it is clear that a single formal notation or method for supporting these viewpoints/structures will be formidable to construct. Additionally, formal models of the architecture will not be suffi cient: certain system attributes may need to be modelled in other ways, notably performance and usability (which are best explored by prototyping). It is likely that a combination of formal notation and informal prose is best suited to the architecting stage. In order to unite and relate the developer's formal, high-level software structure with the expectations of the other stakeholders, an informal rationale is necessary. The rationale should explain the reasoning behind the software structure, and show how it satisfi es the expectations of the customer, user, and maintainer. However, even the informal rationale needs careful defi nition of terms and concepts, to avoid costly misunderstandings among the stakeholders.

6. Conclusion

We have presented a definition of " software system architecture" which provides a set of criteria for determining whether a given complex of diagrams and symbols is an architecture or not. It is based on making the architectural rationale a first-class citizen in the definition, and on requiring the rationale to ensure that the architecture's components, connections, and constraints define a system that will satisfy a set of defined stakeholder needs for the system.

This definition enables the existence of a software system architecture to serve as the key milestone in the system's lifecycle: the decision of whether or not to proceed into full scale development and evolution of the system. For even moderate size systems, if they do not have such an architecture, they should not be built.

Such an architecture provides a process target for the early spiral cycles of system definition, and a guiding framework for the remainder of the life cycle. This definition of " architecture" also has significant implications for architecture definition languages, particularly on their need to support visualization of and reasoning about stakeholder concerns.

7. Bibliography

Alexander, C. (1964), Notes on the Synthesis of Form, Harvard University Press, Cambridge, Ma, 1964.

Binns, P., Englehart M., Jackson M., and Vestal S., *Domain-Specific Software Architectures for Guidance, Navigation, and Control*, Honeywell Technology Center, Minneapolis, Mn. (available at ftp site honeywell.src.com)

Boehm, B.W. and P. Bose (1994), " A collaborative Spiral Software Process Model Based on Theory W," Proceedings, ICSP 3, IEEE, Reston, Va. October 1994.

Boehm, B.W. and W. L. Scherlis (1992), " Megaprogramming," *Proceedings of the DARPA Software Technology Conference*, April 1992. (available via USC Center for Software Engineering, Los Angeles, CA, 90089-0781).

Factor, M., D. Gelernter, C. Kolb, P. Miller, D. Sittig (1991), " Real-Time Data Fusion in the Intensive Care Unit," IEEE Computer, vol 24, no 11, November 1991, pp. 45-53.

Gacek, C., A. Abd-Allah, B.K. Clark, and B.W. Boehm (1994), " Focused Workshop on Software Architectures: Issue Paper," *Proceedings of the USC-CSE Focused Workshop on Software Architectures*, June 1994.

Garlan, D. and M. Shaw (1993), " An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific Publishing Company, 1993.

Hayes-Roth, F. (1994), *Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program*. Teknowledge Federal Systems. Version 1.01, February 1994. (available from Teknowledge)

Jones, A. K. (1993), "The Maturing of Software Architecture," Software Engineering Symposium, Software Engineering Institute, Pittsburgh, Pa., August 1994.

Kruchten, P. (1994), "An Architectural Model for Large-Scale, Distributed, Software-Intensive Systems," *Proceedings of the USC-CSE Focused Workshop on Software Architectures*, June 1994.

Luckham, D., Augustin L., Kenney J., Vera J., Bryan D., and Mann W. (1994), *Specification and Analysis of System Architecture Using Rapide*, 1994. (available at ftp site anna.stanford.usc.edu)

Penedo, M. H., and W. Riddle (1993), "Process-sensitive SEE Architecture (PSEEA)--Workshop Summary", *Software Engineering Notes*, ACM SIGSOFT, vol. 18, no. 3, July 1993, pp. A78-A94.

Perry, D.E. and A. L. Wolf (1992), "Foundations for the Study of Software Architecture", *Software Engineering Notes*, ACM SIGSOFT, vol. 17, no. 4, October 1992, pp. 40-52.

Terry, A., G. Papanagopoulos, M. Devito, N. Coleman, and L. Erman (1993), *An Annotated Repository Schema*, Teknowledge Federal Systems, Version 3.0, Working Draft, October 1993.

Tracz, W. (1994), "Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)", *Software Engineering Notes*, ACM SIGSOFT, vol. 19, no. 2, April 1994, pp. 52-56.