1995

Summary of the '95 Monterey Workshop - Specification-Based Software Architectures

Berzins, V.; Shing, M.

https://hdl.handle.net/10945/46119

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun
Summary of the ‘95 Monterey Workshop - Specification-Based Software Architectures

V. Berzins and M. Shing

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1. Context

The focus of the 1995 Monterey Workshop was on the relation among specification-based software architectures, formal methods, and practical tools for software development. The study of software architectures is a young and evolving field. Although the term "software architecture" is not explicitly mentioned, some early work by Wiederhold et al proposes a component-based software technology for programming in the large, where software systems are made up of subsystems (called megamodules) glued together by megaprograms. The major difference between megamodules and traditional modules is that the former "encapsulate not only procedures and data, but also types, concurrency, knowledge, and ontology"[6].

DeMarco likens "the architecture of a complex software system" to "the infrastructure of a highly evolved social system or biological organism. It is a framework for the disciplined introduction of change. The difference between the two is that design, as we commonly use the word, applies to a single product, while architecture applies to a family of products"[2].

Garlan and Perry, in their Guest Editorial for the IEEE TSE Special Issue on Software Architecture, define software architecture as "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time"[3].

Shaw views the architecture of a software system as a way to "define that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, structural and semantic differences among components and interactions"[5].

Boasson, in the Guest Editor’s Introduction to the IEEE Software Special Issue on Software Architecture, uses the term architecture to mean "a system structure that consists of active modules, a mechanism to allow interaction among these modules, and a set of rules that govern the interaction"[1].

According to Berzins and Luqi, software architecture defines "the common structure of a family of systems by identifying and specifying (1) the components that comprise systems in the family, (2) the relationships and interactions between the components, and (3) the rationale for the design decisions embodied in this
information"[4].

While the above definitions differ in the scope which software architecture should cover, there is general agreement that a software architecture addresses a family of related systems and involves a mapping from a problem space into a solution space.

2. Workshop Highlights

The workshop had a series of presentations related to different aspects of specification-based architecture, and extensive discussions. The workshop schedule was organized as follows:

- **Day 1: Tuesday Sep. 12, 1995**
  
  Luqi, Naval Postgraduate School: *Welcome and Introduction*
  
  Waugh, SEI/CMU: *Evolutionary Design of Complex Software*
  
  DeMarco, Atlantic Systems Guild: *On Systems Architecture*
  
  Tsai, Univ. of Illinois at Chicago: *A Knowledge Based Approach for Specification-Based Software Architectures*

- **Day 2: Wednesday Sep. 13, 1995**
  
  Berzins, Naval Postgraduate School: *Software Architectures in Computer-Aided Prototyping*
  
  Goguen, Oxford University: *Algebraic Specification-Based Software Architectures*
  
  Dampier, Army Research Laboratory: *Specification Merging for Software Architectures*
  
  Clements, SEI/CMU: *Formal Methods in Describing Architectures*

- **Day 3: Thursday Sep. 14, 1995**
  
  Mok, Univ. of Texas at Austin: *Real Time Aspects of Software Architecture*
  
  Robertson, University of Edinburgh: *Lightweight Formal Methods*
  
  Cooke, Univ. of Texas at El Paso: *The Software Architecture for the Analysis of Geographic and Remotely Sensed Data*
  
  Berzins, Naval Postgraduate School: *Workshop Conclusion*

Day 1 began with Luqi’s opening remarks, which focused the attendees on the importance of software architectures and the relation between software architectures and other issues in software engineering. The importance was underscored by Waugh’s presentation of the new ARPA SISTO program to enable rapid evolution of system software. DeMarco then commented on the cost of good software architecture: “system architecture is expensive, but probably not as expensive as its absence”, and concluded that “the problems that have most often hampered architectural efforts have not been technical, but political, economic and sociological”. The day ended with
Tsai’s presentation on a framework for constructing a specification-based software architecture using a frame-and-rule oriented requirements specification language FRORL.

Day 2 started with Berzins’ presentation on software architectures in computer-aided prototyping, which examined representation and support for software architectures in computer-aided prototyping. The talk also explored the connection between generic software architectures and automation support for software reuse, program generation, software evolution, reengineering, and transformation of prototypes into production code. Goguen presented parameterized programming and module expression as a means for developing specification-based architecture. A module expression describes the architecture of a system as an interconnection of component modules, and executing the expression actually builds the system. Dampier explored the view that software prototypes are architectural representations for the intended software system and presented specification merging as a method of aiding in the evolution of software prototype. Day 2 concluded with Clements’ presentation on Modechart and its analysis environment for specifying hard-real-time embedded computer systems.

Day 3 began with Mok’s talk on the use of real-time logic for specifying, analyzing, and synthesizing hard real-time system architectures, followed by Robertson’s presentation on a collection of “lightweight” formal methods and tools, which can easily be picked up and which offer an obvious gain to practitioners after a short training span. These methods and tools described rely on a particular, strongly reinforced, style of specification architecture. Cooke discussed a software architecture to support the analysis of earth data, which involves the use of a wide range of sophisticated software tools and exploratory programming languages, and the need for a formal specification language to describe the functionality of the various tools which make up the architecture. Day 3 ended with the open discussions on lessons learned during the workshop.

3. Workshop Summary and Conclusions

This section summarizes and synthesizes the conclusions reached in discussions during the three day workshop.

3.1. What is Software Architecture

A number of definitions of software architecture were presented, and they all agreed that software architectures include system components and system structure. A substantial fraction of the definitions also included rationale as part of the architecture.

Traditionally a system architecture identifies each system component, specifies what kind of component it is, and describes its attributes. This view was refined at the workshop as follows:

1. A component should be an encapsulated subsystem. An encapsulated subsystem has a definite boundary, its observable behavior consists of the interactions that cross that boundary, and those interactions conform to a specified interface. The attributes specified in the architecture should be those visible from the outside of the subsystem. In particular, a specification of the observable behavior of the
A subsystem should be part of the architecture and any concrete component satisfying that specification should be a valid filler for the architecture slot.

(2) There is a distinction between the behavior of a particular component and the specification associated with a component slot in an architecture. The component specifications in a software architecture are of the second kind. In different instances of an architecture, the same slot can be filled with different components. However, every component that fills the slot must satisfy the behavioral requirements specified for that slot. Those requirements may constrain the behavior of the components that can fit into the slot without completely determining it. For example, a lexical scanner component in a compiler architecture must generate the sequence of tokens in the source text; different instances of the architecture can process different languages, with different definitions of what constitutes a token. The behavioral variations allowed by the constraints associated with an architecture slot are important because they determine the size of the system family defined by the architecture, and the degree to which the architecture supports system evolution.

A common view is that an architecture specifies the structure of a system. This was refined at the workshop as follows:

(1) Structure is a multi-dimensional idea; there are many kinds of structure and a single system can have more than one kind of structure. Some of these include (a) decomposition into subsystems, (b) restrictions on interactions between subsystems, (c) generalization/specialization relations among components (e.g. subclass hierarchies), (d) packaging structure (e.g. groupings associated with files, tasks, processors), and (e) timing/scheduling constraints.

(2) A software architecture includes both a black-box view of the entire system and information about its subsystems and how they interact. Specification languages have traditionally focused on the first half of this; an architecture description language should have capabilities for defining system structure in addition to the capabilities traditionally associated with specification languages.

(3) The description of the interactions between subsystems depends on the underlying model of computation. Ideally the description should be abstract and independent of programming language. However, the nature of the components and the relationships between them may depend on the computation model. Most current treatments of software architecture are implicitly based on an imperative computational model; functional models seem to be subsets of this and are not radically different. However, computational styles such as attribute grammars and logic programming may require a different set of primitives to describe their structure, especially if we are aiming for the simplicity that comes with a high level of abstraction. This area is mostly unexplored at the current time.

(4) Concrete interfaces and low-level packaging aspects that are dependent on programming language (e.g. function vs. procedure interfaces) are also part of the architecture. However, they should be kept separate from the more abstract aspects, and treated as refinements. Ideally the concrete refinements should be optional (i.e. they should have default values, or be automatically derived from
the particular components used to fill component slots in the architecture).

3.2. How Software Architecture Can Be Used

The expected benefits of architecture include: (1) aid / speedup / reduced cost in constructing solutions, (2) management of quality attributes, such as changeability or performance, (3) more effective evolution, (4) prediction of final system behavior, (5) improved reliability, and (6) support for simple "niche languages", which achieve simplicity because some knowledge of the problem domain is built into the language and supporting tools. These benefits are expected to come about because of more effective software reuse, automated program generation, and reduced system integration problems due to the consistent conceptual models provided by the architecture.

Evolution, both in the contexts of prototyping and deployed software, is a form of "navigation through the problem domain covered by an architecture". If the architecture can be general enough so that evolution stays within the confines of the covered part of the problem domain, this is obvious. Some of the work presented at the workshop shows that some automation support for the evolution of the architecture itself is also possible. An important kind of architectural evolution is generalization, which extends the family of systems addressed by the architecture.

The increase in reliability comes from reductions in conceptual incompatibilities in large systems and more effective reuse of well-tested designs. Improved analysis capabilities also play a role. For example, a method for providing automation support for timing feasibility analysis was presented at the workshop.

3.3. How Has Software Architecture Been Used In Industry

Tom DeMarco contributed some sobering insights into commercial attitudes on software architecture. No major commercial application of software architectures has made a profit so far, so companies are understandably reluctant to invest anything in architectures. This has been due to anti-trust actions by the US Treasury Department rather than to any fundamental technical or economic issues. Cases cited to support this include Xerox PARC, AT&T, and IBM. Talligent was cited as an exception, and it was noted that the Treasury Department is considering anti-trust action in this case also. Some limiting factors on investment in architecture are: (1) up-front cost and associated risk of not recovering it (an architecture is several times more expensive than an instance), (2) time (often really cost limits in disguise), and (3) lack of accepted criteria for distinguishing good architectures from bad ones.

3.4. Directions for Future Research

Some directions for future research suggested by the discussions at the workshop include

(1) Using software architectures to provide automated support for software evolution, and

(2) Investigating methods for effectively representing design rationale so that it can be used to provide automated decision support. Some issues mentioned were how to capture design knowledge and how to model design decisions.
4. References


