To appear in Proceedings of 8th International Workshop on Software Specificatin and Design, March 1996

Truth vs Knowledge: The Difference Between What a Component Does and What We *Know* It Does

Mary Shaw
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

Conventional doctrine holds that specifications are sufficient, complete, static, and homogeneous. For system-level specifications, especially for software architectures, conventional doctrine often fails to hold. This can happen when properties other than functionality are critical, when not all properties of interest can be identified in advance, or when the specifications are expensive to create. That is, the conventional doctrine often fails for practical software components. Specifications for real software must be incremental, extensible, and heterogeneous. To support such specifications, our notations and tools must be able to extend and manipulate structured specifications. In the UniCon architecture description language, we introduce ***credentials****, a property-list form of specification that supports evolving heterogeneous specifications and their use with system-building and analysis tools.*

Conventional software doctrine calls for component specifications that are:

- *Sufficient and complete:* the specification of a component says everything a user needs to know or is permitted to rely on about how to use the component,
- *Static:* the specification can be written once and frozen, and
- *Homogeneous:* the specification is written in a single notation.

For example, a typical discussion of the promise of reuse [Ben95] is introduced,[1]

> Three prerequisites must be met for a component to be used in more than one system: complete, opaque enclosure; complete specification of its external interface; and design consistency across all sites of reuse. Without these, reuse will remain an empty promise.

It may be possible to adhere to the conventional doctrine for algorithms and data structures, or when functionality is the only property of interest. However, architectural, or system-level, components cannot in practice satisfy these criteria. Indeed, they inevitably will not, and it is impractical to try.

This paper is about what this implies and how to cope with it: why architectural specifications are insufficient, incomplete, incremental, and heterogeneous -- and how software development methods and tools must adapt in response.

Section 1 describes architectural components and explains why they cannot adhere to the conventional doctrine. Section 2 considers many of the properties that need to be specified. Section 3 sets out requirements for architectural specifications. Section 4 introduces an approach to a solution, *credentials* for those properties that have been specified to date.

1. Architectural components and their specifications

Software architecture deals with the overall structure and properties of software systems. The most common architecture description languages (ADLs) support components, connectors, and other aspects of the system such as styles, constraints, or design rationale [ShawGar96, PerWolf92]. Although the issues raised here apply to all architectural elements, this discussion focuses on specifications of the components, which may be either primitive (i.e., written in a programming language) or composite (i.e., defined in the ADL).

The information required to use an architectural component goes beyond computational functionality to include *structural* properties that affect how the

[1] Note that this description comes from the applications community, not the formal methods community.

component can be composed with other components; *extra-functional* properties that describe performance, capacity, environmental assumptions, and global properties; and *family* properties that assert relations among similar or related components. Software development environments should accommodate an open-ended collection of tools for construction and analysis. Different tools may depend on different properties, and some tools may generate new specification information [ICSE95].

Specifications of architectural components are *intrinsically incomplete* because system correctness depends not only on computational functionality but on other properties as noted above [Shaw85, GAO95]. It's impractical to expect full specifications of all these properties because of the prohibitive effort required to specify a wide variety of properties, whether or not anyone will use the information. Worse, it's impossible: the developer cannot anticipate all the aspects of the component that its users might care about. As an added complication, the degree of precision in the specification may be influenced by the tradeoff between the costs and benefits of improved precision [Shaw81]. Although completeness is impractical, it is still appropriate to expect specifications for a common core of properties, and it is reasonable for a tool to require certain properties. Reasoning with partial specifications has already received some attention [Jac94, Per95].

Specifications of architectural components must be *extensible*, because developers discover new kinds of dependencies as they attempt to reuse independently-developed components together. Even with the best of good faith, component developers cannot describe all the incidental ways their components may interact with the entire environment. Garlan and colleagues analyze the implicit assumptions that interfered with one instance of attempted reuse [GAO95]. Not only is much important information implicit, but users have no effective way to capture information they discover for future reference. As the specifications are extended, information about a property may be received from multiple sources; these must be reconciled [BarWing90].

Specifications of architectural components must be *heterogeneous*, because of the diversity of significant properties, as described in Section 2. It is unreasonable to expect a single notation to serve for all of them.

Thus the drivers of specification incompleteness, extensibility, and heterogeneity are

- *Open-ended needs:* The designer cannot anticipate all properties that may ever be of interest to some user. Further, future users may find new ways to take advantage of old properties. Interesting properties are of many different kinds.
- *Cost of information:* Even for common properties, it is not practical to produce a complete specification. Further, the precision of a specification may be selected to balance the cost of getting a tight bound against how badly it's needed. The cost of understanding a specification also affects its utility.
- *Evolution:* As time passes, new properties may be added to a specification because someone (not necessarily the developer) discovers new information or new dependencies. Developers can often make progress with partial information but take advantage of additional information.

2. Architectural properties

The main reason why architectural components require incomplete, extensible, and heterogeneous specifications is the diversity of facts about a component that may affect a designer's ability to compose it with other components and achieve a correct and consistent result. This section describes three major classes of properties that augment the conventional functional properties of type, signature, and pre/post conditions.

2.1 Structural properties

The most significant properties for architectural design deal with the ways components interact, and hence with the ways those components can be combined into systems. Especially important is the *packaging* of a component, which includes the type of component and the types of interactions it is prepared to support. The choice of packaging is often largely independent of the underlying functionality, but components must be packaged in compatible ways if they are to work together smoothly. For example, unix provides both a sort system call and a sort filter; although they have the same functionality, they are far from interchangeable. Some common packagings for components and the ways they interact are:

Component type	*Common types of interactions*
Module	Procedure call, data sharing

Object	Method invocation (dynamically bound procedure call)
Filter	Data flow
Process	Message passing, remote procedure call, other communication protocols, synchronization
Data file	Read, write
Database	Schema, query language
Document	Shared representation assumptions

Distinctions of this kind are now made informally, often implicitly. If the distinctions were more precise and more explicit, it would be easier to detect and eventually correct incompatibilities by analyzing the system configuration description. Such checking must address not only local compatibility (e.g., do two components expect the same kinds of interactions) but also global properties (e.g., are there loops in a data flow system?).

2.2 *Extra-functional properties*

In addition to functionality and structure, architectural specifications must be capable of expressing extra-functional properties related to performance, capacity, environmental assumptions, and global properties such as reliability and security [MCN92, Shaw85, CBKA95]. Many of these additional properties are qualitative, so they may require different kinds of support from more formal specifications. These other properties include:

- time requirements
- precision and accuracy
- timing variability
- reliability
- real-time response
- robustness
- latency
- security
- bandwidth and throughput
- service capacity (e.g. # of clients/server)
- space requirements
- dependence on specific libraries, services
- space variability
- conformance to an interface standard
- possession of main thread of control
- conformance to implementation standard
- minimum hardware configuration
- intended profile of operation usage

Some of these properties require periodic updates, especially those that assert conformance to external standards (e.g., Windows 95) that may themselves change.

The formal specifications familiar to the IWSSD community are not the most common kind. More prevalent are product descriptions such as the following, which specifies the interface between a software product and the environment required to run it [DeL95]. This specification deals with space and conformance to established standards. The functionality of the product is described (imprecisely) in associated prose and pictures.

IBM or 100% IBM-compatible microcomputer with Intel 80386 microprocessor or higher or 100%-compatible processor.

Minimum 4 MB RAM., 3 MB of available space on a hard disk.

ISO 9660-compatible CD-ROM drive with 640+ MB read capacity and Microsoft® CD-ROM extensions.

Microsoft®Windows˙-compatible printer (not plotter) recommended, with 1.5 MB printer memory for 300 dpi laser printing, 6 MB for 600 dpi.

Microsoft®Windows˙-compatible mouse (recommended).

Microsoft®Windows˙-compatible VGA card and monitor.

Microsoft®Windows˙ version 3.1 and MS-DOS® version 4.01 or later.

2.3 *Family properties*

Components are often designed in families, sharing assumptions about such things as division of responsibilities, data encoding and protocols. A large family of systems may have constraints on collections of components that must be used together. It may also be important for a specification to express not only the properties of the instance at hand, but also a larger envelope of capability that could be achieved by, for example, modifying setup parameters of changing inheritance relations.

3. Requirements for Practical Architectural Specifications

Component specifications play two roles:

- *Implementation:* giving information about "as-built" capabilities of individual existing components ("How do I *use this component*?")
- *Requirement:* setting out the requirements that a component that has not yet been selected or constructed must satisfy ("What component is needed to *fill this hole*?")

These roles are roughly analogous to the formal and actual parameters of procedures--they serve both to define the capability envelope of a required component and the actual capability of an instance. Just as actual and formal parameters of procedures differ in detail, so do the requirement and implementation specifications of components.

Given the setting described in Section 1, specifications require more support than they get at present (static text files in a given formal syntax). Let us consider, then, the requirements for models, methods, and associated tools that support specifications of the sorts of architectural components that appear in Real Life™. That is, what happens if you force the models to adapt to real-world elements rather than vice-versa? Some of the capabilities that must be supported for practical specifications are

- *Tolerate incompleteness*. Analysis tools must be able to indicate which properties they depend on; if information is missing, they must either explain why analysis can't proceed or warn about the limitations of the results. It should be possible to prohibit dependence on a property -- to be "actively silent"
- *Collect specifications incrementally*. Not only developers, but also users, must be able to add information to specifications. The source, and hence the credibility/validity of the information must be preserved.
- *Support specifications of many properties* in different notations. Add new properties as they turn out to be interesting.
- *Propagate new information.* When new information is supplied, it must be propagated to places where it might improve prior analyses. Further, some properties may be derived analytically rather than declared by the designer. These can often be improved with new information.
- *Invalidate specifications* when appropriate. Modifications to a system definition or to the sources of derived information may render individual parts of a specification invalid.
- *Search* for components that partially match a partial specification, with an indication of the goodness of fit. [ZarWing95]
- *Support checking*, both that a component specification and its associated implementation are consistent and that a configuration of components is well-formed. Support tools to make minor adaptations when minor mismatches are detected. Support incremental checking for incremental specification.
- *Support flexibility*. Define limits on actual values of properties; describe the envelope of allowable behavior (retaining information about both the envelope and the current instance); separate policy from mechanism.
- *Yield partial value for partial information*, incremental value for incremental information.

4. Credentials for What We Know Is True

To address this problem, we propose the notion of *credentials*: incremental, evolving specifications. Credentials may be viewed as property lists, or lists of <attribute, value> pairs. Credentials must include

- *registered attribute names* and provisions for adding new names (including private ones); a means of indicating which ones are required or optional under certain circumstances
- *multiple notations* for values of attributes
- *credibility*, or sources for the values of attributes. They might, for example, include

asserted:	given by designer, taken on faith
verified:	proposed by designer, verified by tool
derived:	derived (preferably automatically) from other specifications
default:	provided as part of component definition
forced:	determined by nature of definition (for example as part of subtyping definition)

Credentials must be an integral part of the software definition, so that they are supported by the CASE environment and updated in tandem with the code. The associated tools must support operations including

- compatibility checks similar to type checking but involving a richer set of properties
- access to externally-defined tools, including extraction of relevant attributes for use by the

- tool and incorporation of results from the tools as new attribute values
- rules for resolving values for a given attribute that are proffered by multiple sources
- invocation of analysis for checking credentials after code is modified, including invalidation of properties whose values can no longer be confirmed

The UniCon architecture description language [SDKRYZ95] supports the bare bones of this proposal. UniCon specifications are given in the form of property lists; the set of attributes is open-ended; and particular attributes are required for certain checks and tools. Current development will add credibility values for attributes and make explicit the set of notations (including "uninterpreted") for values of attributes.

Research Support

The work reported here has been heavily influenced by an ongoing collaboration with the Composable Systems Group at Carnegie Mellon University, particularly on discussions with David Garlan, Greg Zelesnik, and Rob DeLine. It draws on material presented at the 1995 Dagstuhl in Software Architecture and in a special volume of Lecture Notes in Computer Science. It was improved during discussions in the Software Architecture Reading Group at CMU. It was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency, under grant F33615-93-1-1330 and by a grant from Siemens Corporation. It represents the views of the author and not of Carnegie Mellon University or any of the sponsoring institutions.

References

[BarWing90] Mario R. Barbacci and Jeannette M. Wing. A language for distributed applications. *Proc 1990 Int'l Conf on Computer Languages*, pp. 59-68.

[Ben95] Douglas W. Bennett. The promise of reuse. *Object Magazine*, vol 4, no 8, January 1995, pp. 32-40.

[CBKA95] Paul Clements, Len Bass, Rick Kazman, and Gregory Abowd. Predicting software Quality by architecture-level evaluation. In *Proc Fifth International Conf on Software Quality*, October 1995

[DeL95] DeLorme Mapping Company. WWW page describing MapExpert product. URL: http://www.delorme.com/catalog/mex.htm, 1995.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch, or Why it's hard to build systems out existing parts. *Proc 17th International Conf on Software Engineering (ICSE-17),* April 1995.

[ICSE95] David Garlan. *Report of ICSE-17 Software Architecture Workshop*, to appear ACM SIGSOFT Software Engineering Notes, 1995.

[Jac94] Daniel Jackson. *Structuring Z Specifications with Views*. Carnegie Mellon University Technical Report CMU-CS-94-126.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering*, vol 18, no 6, June 1992.

[Per95] Dewayne E. Perry. System Compositions and Shared Dependencies. Unpublished manuscript, January 1995.

[PerWolf92] Dewayne E. Perry and Alexander L. Wolf, Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, vol 17, no 4, pp. 40-52, October 1992.

[Shaw81] Mary Shaw. When Is 'Good' Enough?: Evaluating and Selecting Software Metrics. In *Software Metrics: An Analysis and Evaluation,* A. Perlis, F. Sayward, and M. Shaw (eds), MIT Press, 1981, pp. 251-262.

[Shaw 85] Mary Shaw. What Can We Specify? Questions in the domains of software specifications. In *Proc Third International Workshop on Software Specification and Design*, pp. 214-215, August 1985.

[ShawGar96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SDKRYZ95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Tr on Software Engineering*, May 1995.

[ZarWing95] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components." *Proc. of SIGSOFT Foundations of Software Engineering*, October 1995.

Keywords: software architecture, specification, software analysis, extra-functional properties, incremental specification, credentials