Improving Architectural Description Languages to Support Analysis Better

Lori A. Clarke
Department of Computer Science
University of Massachusetts
Amherst, MA 01002
clarke@cs.umass.edu

ABSTRACT

Software architecture is an emerging new area. It encompasses the traditional area of software design with an emphasis on the design of distributed systems. Based on the past failures of complex design techniques to gain wide spread acceptance, software architecture languages must demonstrate that they provide significant value. A primary benefit will be the early detection of software faults. This is particularly important for distributed systems, since even simple distributed systems can be difficult to understand. This paper outlines some ways in which architectural description languages need to be designed to increase their usability, acceptability, and consequently analyzability.

KEYWORDS
software architecture, distributed systems, software analysis

On the one hand, software architecture is nothing more than the renaming of a well-established research area, namely software design. On the other hand, this renaming is perhaps justified since software systems are very different from how they use to be and new software design techniques need to be developed to address those differences. The primary differences are that systems now utilize an object oriented, polymorphic typing model that encourages reuse and that systems are distributed collections of subsystems. The first change affects how systems are represented and has been the focus of several design modeling notations such as OMG and UML. The second area is the major concern of software architecture; that is, how to design distributed software systems. It is this second area and the importance of analysis that is addressed in this position paper.

Distributed systems have many advantages, such as concurrent execution, with its concomitant potential for great savings in execution time, and the flexibility of component substitution. Unfortunately, such systems are much harder to reason about, and thus much harder to design correctly. Since it costs less to detect and fix flaws in a system early as opposed to later in the development process, it is advantageous to develop design notations for distributed systems that will lay the foundation for analyzing those designs and detecting problems early.

In addition to analyzability, we know from past experience that design notations must be simple and visually appealing. They are intended to convey information about the system relatively easily and thus are provided to facilitate system understanding. We also know from past experience with numerous graphical program representations that such representations make great demonstrations, but few developers use them in practice. As soon as systems get large or complex, a visual representation is not that helpful. For example, flow charts and control flow graphs are not widely used by system developers as software understanding aids; they are used as the basis for underlying analysis however.

Many of the current architecture description languages fail to satisfy these two goals. Some, such as Rapide, are basically supercharged programming languages. Although visually appealing graphical representations can be provided, users must be familiar with the various architecture programming notations that comprise Rapide. Other languages, such as Wright, are based on mathematical formal models; such notations have never been widely accepted in industry and thus will probably not be widely embraced outside the academic community. Others, such as ACME, are graphical notations that are relatively easy to view but for which there is no agreed upon semantics upon which to base interesting and useful analysis. Thus, if accepted by practitioners, there will be numerous dialects of ACME, thereby increasing the confusion and hampering software understanding.

In this paper I advocate four requirements for architectural description languages that focus on how architectural description languages can better support analysis as well as software understanding.

CONNECTOR ABSTRACTION:

Architecture description languages must provide support for the most common component interaction abstractions. When procedure invocation was the primary means of component interaction, design language notations easily represented this control construct (usually by arrows). Now, however, there are several commonly used interaction models for distributed systems, such as remote procedure call, message passing, and event based notification. These interaction abstractions must be first class entities in the architecture description language. While it might still be useful to have a generic connector that can be instantiated in terms of one of these higher level distributed control constructs, it is also important that the choice of interaction can be stated explicitly and easily.

Analysis techniques have tended to focus on the synchronous communication model that is used in Ada and CSP. Until recently, this was the only reasonable alternative; Ada was one of the few languages that had a well-defined concurrency control construct. C and C++ in contrast tended to use platform dependent constructs. Developers of analysis techniques must focus on general and widely used models. It is too expensive, in terms of human effort, to develop approaches that address one-of-a-kind techniques.

Architecture researchers and analysis researchers should join forces to help define a reasonable set of connector constructs. These constructs should represent the main concurrency control mechanism in use today. If architectural description languages supported these constructs, these languages would have a greater chance of being widely used, and analysis researchers would have a test bed of examples on which to evaluate their analysis techniques. As new constructs are developed and gain acceptance, these too could be added to the library of abstract connectors. Considering past history, many such constructs might be proposed, but few would gain widespread acceptance, and thus this library would grow very slowly. With appropriate forethought, analysis techniques could even be developed that support multiple concurrency control abstractions, thereby being applicable to the many software systems with mixed distribution paradigms.

ARCHITECTURAL STYLE:

The early work on software architecture by Mary Shaw advocated recognizing common templates and learning how to exploit that information. It is interesting that this approach, now called design patterns, has become very popular in programming. Programmers recognize that they are frequently reinventing the wheel and that it behooves them to be knowledgeable about these patterns. As with many breakthroughs in software development, patterns bring a higher level of abstraction to the software development process. Earlier work focused more on capturing abstractions directly in the programming language, as was the case with structured control constructs (e.g., case statements), data abstraction, and object oriented programming constructs. Design patterns use the primitive constructs in a language to capture higher level abstractions. Previously this had been done with data structures, such as stacks and queues, but with patterns the focus is on control abstractions instead of data abstractions.

Software architect researchers need to explore and define common architectural control abstractions. Some of these have been identified, such as the pipe and filter style and the heartbeat style, but there still is not wide spread acceptance of these styles nor agreement on the requirements or constraints that identify such a style.

These architectural control abstractions along with their specification requirements will provide considerable opportunities for analysis. If a developer believes that an architectural description should satisfy a pattern then the associated specifications for that pattern can be validated by analysis techniques. Since writing specifications is a difficult and error prone process, having generic specifications associated with well-known and accepted patterns would help considerably. These would be application independent, style specific properties. As noted below this would also help with compositional development and software understanding.

COMPOSITION AND REFINEMENT:

Software architecture is usually concerned with supporting the development of large systems. Such systems are not developed all at once, but are incrementally evolved over time. Thus, architectural description languages and associated analysis techniques should support incremental development. There are several aspects to this problem.

One of these is supporting the description and analysis of incomplete systems. It is useful if missing components can be represented by high-level specifications. These specifications are assumed to be true during the analysis of the existing components. When a missing component becomes available, any specifications that were assumed about this component would need to be validated. If found false, any previous analysis that was based on these false assumptions would have to be revisited.

Such an approach supports another aspect of incremental development--that is, the incremental development of specifications of such systems. With such an approach, analysis is undertaken hand-in-hand with development and, thus, developers receive early feedback as the system evolves. Specifications of missing components would be incrementally developed depending on the contexts in which each is used. When a new component is defined, it is checked to be sure that it is consistent with all these contexts. If not, it is clear what re-analysis and redesign must be considered. In addition, any new specifications of the component would be defined and validated.

A SIMPLE FORMAL SEMANTICS:

Architectural description languages need to be relatively simple to create, understand and maintain or developers will eschew their use. Even if analysis of such descriptions is shown to reduce errors and decrease overall development and maintenance costs, these languages will not be used by developers unless they can see short term benefits in their use. This pessimistic view is validated by studies of software inspection techniques, where it has been shown that even this well-known effective technique (of which there are very few) is dropped when resources are tight. The more pain (i.e., intellectual effort) required to use a technique, the more likely it will be avoided. Thus, architectural specification must be relatively easy to describe, must provide relatively appealing visual representations, and must be shown to be effective.

Analysis of architectural representations is the primary means by which this approach will be shown to be effective. If errors are caught and corrected early, then overall develop costs will be reduced. However, analysis can not be done unless the languages have meaningful semantics upon which to base this analysis and unless developers use these architectural description languages. This would seem to be a contradiction since developers often shun formal, semantically rich models.

Recent work on specification patterns by Dwyer, Avrunin and Corbet is addressing some of these concerns. This work has been developing a high-level, natural language like specification notation. This notation can be mapped unto many of the more mathematically formal and thus obtuse specification languages upon which many analysis techniques rely. Thus developers do not need to understand or use the more formal notation. A similar approach needs to be developed for architectural specification languages. Just like programmers no longer need to understand the low-level register operations that underlying a high-level programming language, specifications and architectural description languages should relieve developers of understanding the precise, formal semantics that must underlie these languages.

In summary, architectural description languages have the potential to improve the quality of software systems and to reduce software development effort. These benefits are based on the assumption that powerful analysis techniques will be developed to support analysis of system descriptions written in these languages. Without such support, architectural descriptions will not provide enough benefit to warrant the cost of their development. Thus analysis and software architecture researchers must work together to address the needs of developers. There is great potential to improve the software development process but researchers must be pragmatic about what might work and must develop approaches that can be demonstrated to be effective, that are relatively easy to use, and that take into account current programming practices.