The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective

R. A. Riemenschneider and Victoria Stavridou
Dependable System Architectures Group
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Abstract

Most research on architecture description languages (ADLs) has focussed on the use of architecture description to guide conventional, top-down development of systems. This paper describes one approach to bringing the benefits of formal architecture description to the world of component-based development (CBD), where system developement consists primarily of connecting instances of pre-existing components.

1. Using Architecture Descriptions to Achieve Dependability

The principal focus of our group's ADL work has been on describing architecture implementation patterns (i.e., rules for implementing abstract architectural constructs in terms of more concrete constructs), developing techniques for proving that patterns preserve dependability properties of interest (e.g., security, fault-tolerance), and using those patterns to incrementally generate implementation-level descriptions of architectures from abstract, easily analyzable architectural descriptions. For example, we have developed a reference implementation of an architecture for secure distributed transaction processing (SDTP) -- an extension of X/Open's DTP architecture -- and proven it secure by showing that an abstract description of the architecture is secure and that the patterns used to generate the implementation of the architecture preserve security. (See M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong, ‘‘Secure software architectures’’,*Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 84-98, and F. Gilham, R. A. Riemenschneider, and V. Stavridou, ‘‘Secure Interoperation of Secure Distributed Databases’’, to appear.)

In our view, the scope of architectural description is quite broad. We consider any description of a software system in terms of *components*, joined by *connectors*, that satisfy a collection of *constraints* to be an architectural description. Components correspond to the ‘‘boxes’’ of the ubiquitous ‘‘boxes and arrows’’ diagrams used for informal architecture description. Connectors correspond to the ‘‘arrows’’. What distinguishes architectural descriptions from other sorts of system description is that only the externally visible interfaces of components are described; the details of the internal functioning of components is left unspecified. The architectural description of the system characterizes the ‘‘glue’’ that binds the components together. An architectural description and descriptions of the components jointly constitute a complete system description.

In contrast to our broad sense of architectural description, some members of the ADL community consider a high level of abstraction to be essential to architectural description. They treat architectural description as the first phase of system design, a basis for more detailed lower level designs. This difference in viewpoint is not merely terminological. The broad view encourages the use of ADLs to describe systems, and parts of systems, in terms of connected components at all levels of abstraction, from highest to lowest. The architectural components in a low level description can naturally be identified with the reusable components of the CBD paradigm, providing a link between ADLs and CBD. Our group has recently focussed on ways of strengthening this link, by adapting our ADL toolset to better support CBD, with a goal of providing technology for building dependable systems from off-the-shelf components.

2. Predictably Dependable Computing with Off-the-Shelf Components

One of the challenges in applying the component-based development (CBD) paradigm to critical software systems is guaranteeing dependability. (A *dependable* system is one that can be justifiably relied upon to provide its services.) If a component is nothing more than executable code, how can a developer gain confidence that it will satisfy his requirements? We are currently investigating one approach to answering this question.

In CBD, the developer’s task is to assemble a (large) software system from (a large number of) reusable components, and perhaps a comparatively small amount of code written for the particular application. In general, the developer does not have access to the source code for the components. We shall assume, however, that each component is accompanied by a *specification*. At a minimum, this specification describes the component’s interface, its externally observable behavior, and any constraints that must be satisfied in order for it to behave as specified. The specification is expressed in a formal notation designed for the purpose.

Such specifications can serve several functions. First, they provide a way for the component developer to advertise the component’s capabilities to potential users. Second, they can support sophisticated component search and retrieval capabilities. (See, e.g., A. M. Zaremski and J. M. Wing, ‘‘Specification matching of software components’’, *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT ’95)*, pp. 6-19.) Third, the behavior of the system can be determined from the specified behavior of the components -- assuming, of course, that the components satisfy their specifications -- together with a description of how the components are connected.

From a practical point of view, a system specification constructed by simply conjoining the specifications of a large number of components is not sufficient for analysis purposes. The complexity of such specifications are far beyond the bounds of intellectual tractability. In addition, tools used to establish dependability properties -- such as theorem provers and model checkers -- are equally incapable of dealing with such complex specifications. Simpler, more abstract, descriptions are required by humans and CASE tools alike.

3. Generating Architecture Descriptions by Abstraction

The key observation in adapting out ADL tools to support CBD was that the same architecture implementation patterns we have been using to generate concrete descriptions from abstract descriptions can equally well be used to generate abstract descriptions from concrete descriptions. Our patterns can be thought of as having the form

- If an abstract architectural description matches template T
- And it satisfies the constraints C_1, C_2, ..., C_n,
- Then it can be implemented by the concrete description that matches template U,
- Provided that the concrete description satisfies the constraints D_1, D_2, ..., D_m.

(perhaps together with a collection of ‘‘soft’’ constraints that determine when it is advisable to apply the pattern). A pattern can be used for refinement by matching an architecture against T, checking C_1, C_2, ..., C_n, generating a more concrete architecture that matches U, and adding D_1, D_2, ..., D_m to the constraints that must be maintained on further refinement. But the same pattern can be used for abstraction by matching against U, checking D_1, D_2, ..., D_m, generating a match for T, and subsequently maintaining C_1, C_2, ..., C_n. Consider, for example, a pattern that says an abstract component can be replaced by an interconnected collection of concrete components, provided the public interface is maintained. This pattern supports the familiar sort of ‘‘bubble decomposition’’ (i.e., horizontal component refinement) common to all refinement-oriented development methodologies. However, it equally supports treating a collection of components as a single component at a higher level of abstraction.

Generation of abstract architectural descriptions from concrete descriptions resembles reverse engineering of systems from source code. In terms of the ‘‘bubble decomposition’’ example, generating the abstraction requires identifying a group of components that can usefully be treated as a single abstract component for purposes of some dependability analysis, much as reverse engineering requires identifying pieces of code that can usefully be treated as modules. The problem of module identification based on analysis of source code is known to be extremely difficult. Is there any reason to hope that architecture-based abstraction will be more tractable? We believe that there is. First, since even low-level architectural descriptions are much simpler than source code, because the internals of the components have already been abstracted away. Second, the behavioral descriptions of the connectors are highly structured. Third, our libraries of architectural implementation patterns are ‘‘domain-specific’’. Our belief is that development organizations that reuse components will tend to reuse them in similar ways, so that abstraction patterns that have proven useful in past analyses can be reapplied in similar ways in present analyses.

We are currently working on a demonstration of the feasibility of this approach to generating abstract architectural descriptions from a description of the system as a collection of concrete components (in the CBD sense) linked by concrete connections (e.g., via CORBA). Abstractions in several different ADLs will be generated, each will be analyzed using the tools provided by the ADL, and the pattern instances linking the abstract descriptions to the concrete implementation will be shown to preserve the results of the analysis (exactly as in the case of refinement).

4. Conclusions

ADLs have been shown to provide valuable system analysis capabilities, especially when high levels of dependability are required. But the problem of how to make effective use of these capabilities when systems are constructed from reusable components -- especially when development is principally bottom-up, rather than top-down -- has not been adequately addressed. We believe that out approach to generating abstract architectural descriptions from concrete component-based descriptions can bring the benefits of architectural analysis to the world of CBD.