Architectural Issues in Component-Based Software Engineering *

Michael D. Rice
Computer Science Group
Department of Mathematics
Wesleyan University
Middletown, CT 06459
Tel: (860) 685-2184
Fax: (860) 685-2571
Email: mrice@uts.cs.wesleyan.edu

Stephen B. Seidman
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Tel: (970) 491-5862
Fax: (970) 491-2466
Email: seidman@cs.colostate.edu

Abstract

The architectural style description language *ASDL* can be used to address important issues in component-based software engineering. *ASDL* is built on a collection of software types that support the description of execution and interface semantics for components and systems and the representation of relationships between multiple logical views of systems. It also includes operations for modifying the system state described by the software types. The language provides a domain-independent platform for highlighting and investigating issues that are common to the design of domain-specific component-based systems.

Keywords: Component-based software engineering, software architectural style, software architecture, view

Workshop Goals: Learning, networking, discussing issues related to architectural aspects of component-based software engineering

*This work is supported in part by a grant from the US Naval Research Laboratory.

1 Background

The field of software architectures deals with the design, construction, maintenance, and structure of large software systems, and an increasing proportion of such systems are constructed by assembling existing software components. Architectural issues can therefore be expected to play a significant role in component-based software engineering. The study of component-based software design raises problems analogous to those which motivated the development of data structures, with components playing the role of data types. First, finding useful architectural principles for assembling components is analogous to finding ways in which data structures such as lists, stacks, and trees can be formed from the underlying data types. The importance of identifying such "architectural styles" is now widely recognized [AAG95]. Second, devising architectural description notations that are appropriate for component-based software development is analogous to developing high level programming languages for describing and using data structures. This critical issue has received far less attention. Until recently, most research efforts have developed component-based software systems for building architectures within a specific domain, rather than providing a domain-independent architectural framework for component-based software design. While some current research projects have adopted a more general approach, they are generally either informal [Del96, BBK$^+$97] or have not introduced sufficiently powerful abstractions [Wil96]. In our view, a notion of "abstract software types" can provide a formal basis for describing component-oriented architectural styles, just as abstract data types do for data structures. These software types serve as the foundation for a domain-independent approach to comparing different ways of constructing software systems from components. We have recently developed a formalism and methodology that uses software types to describe software architectural styles. Our approach is summarized in an architectural style description language called *ASDL* [RS96b] that has been used as the basis for a description of the Naval Research Laboratory's *PGM* [KS95, RS96a] component-based signal processing software architectural style.

2 Position

2.1 Overview of ASDL

This section provides an overview of the *Z* schemas [Spi89] that represent the software types and operations of *ASDL*. These types and operations embody our perspective on the essential features of component-based software design methodologies, and the overview therefore corresponds to an outline of our position. A more detailed description can be found in [RS96b]. Examples are given to show how this perspective can be used to specify several important aspects of styles that are associated with component-based software development methodologies, including (a) interface and execution semantics, (b) software system evolution, and (c) relationships between different views of a software system.

2.1.1 MIL Types

In previous research [RS94], we have shown how the essential features of a module interconnection language can be specified with *Z* schemas. These schemas have been incorporated into *ASDL* with only minor changes. The corresponding *MIL* types specify the syntax and static semantics of component interfaces and system modules, respectively. The first (*Library*) schema specifies a library type that provides a collection of templates that describe the interfaces of computational components. A template consists of ports that are used for sending and receiving data; each port has a set of attributes that represent the direction of data movement, the type of the data, and application-dependent information. Some templates are identified as *primitive templates*; these correspond to component interfaces that have been preloaded into the library. Others represent interfaces of *encapsulated modules* that have been placed in the library.

The second (*Setting*) schema specifies a *module type* based on the library type. It consists of a set of *nodes* and a set of *connections* that are defined by *shared labels*. The nodes represent computational components that are instantiated from library templates. The ports on an instantiated node are called *slots*. A label shared by two or more slots creates a connection that can be used for data movement between the corresponding nodes.

2.1.2 Library and Module Types

In *ASDL*, the *MIL* types have been extended to support the specification of execution semantics. The templates in the extended *Library* schema have two additional features: *object attributes* and *semantic interpretations*. The new schema can be thought of as a *semantic library type*. The object attributes are specified by a *part* mapping that provides application-dependent information. The semantic interpretations are specified by an *interp* mapping that associates a *CSP* [Hoa85] process with each template. For example, if a template τ has two ports p and q with direction attributes *in* and *out*, respectively, then the *CSP* process $interp(\tau) = {*}(p?x \rightarrow q!x \rightarrow Skip)$ specifies that the template behaves as a non-terminating copy operation.

The extended *Setting* schema contains a *composition expression* that specifies how the components corresponding to the nodes in a module are executed, and a *semantic description* mapping that assigns a semantic abbreviation to each label used in a module. A composition expression is a restricted type of *CSP* process in which node names are viewed as processes. For example, it may specify that the components will be executed in parallel. A semantic abbreviation associated with a label represents a communication protocol and may include application-dependent information, such as data transfer rates and timing requirements. For example, it may specify bidirectional asynchronous communication and a maximum buffer size. The execution semantics of a module are derived from the semantic interpretations of the templates underlying the nodes, the composition expression, and the semantic descriptions of the labels that specify the connections between nodes. The extended *Setting* schema can therefore be thought of as a *semantic module type* that describes the components of a module and contains the information needed to execute the module. The following examples illustrate the use of library and module types.

(a) The *Processing Graph Method* (*PGM*) [KS95, RS96a] is a component-based software development methodology developed by the US Naval Research Laboratory for signal processing applications. For *PGM*, a module type corresponds to a graph whose nodes correspond to components that perform signal processing operations. The primitive templates underlying the nodes belong to two categories: *transitions* that represent computations and data restructuring operations and *places* that represent data transfers between transitions. Slots of the same port type on nodes instantiated from templates of different category may be connected by directed arcs that indicate data movement.

ASDL can be used to describe the semantics associated with the interfaces of *PGM* nodes. For example, $part(\tau) = G\_Var(T)$ specifies that τ is a place template (specifically, a rewritable *graph variable* that can hold a single token of type T with one input (INPUT) and one output (OUTPUT) port. Its interpretation is a *CSP* process

$$interp(\tau)(R,S) = {}^{*}(\Box\{\mathrm{INPUT}_r?x \rightarrow Skip : r \in R\})\Box(\Box\{\mathrm{OUTPUT}_s!x \rightarrow Skip : s \in S\}))$$

that specifies that τ is willing to receive or send data on the channels that correspond to links to ports on other nodes. The sets R and S index the nodes that are linked to the INPUT and OUTPUT port, respectively.

The interpretation of a transition template enforces *PGM*'s dataflow methodology, in which the execution of a node instantiated from the template is triggered by the arrival of sufficient data at its input slots. The composition expression specifies the concurrent execution of the components corresponding to the nodes in a module and the semantic description of each label specifies unidirectional synchronous transfer of data.

(b) The composition expression can also describe the dynamic evolution of a module within a component-based development environment. In this case, the alphabet of the process representing the expression includes *special event names* corresponding to the *ASDL* operations (see 2.1.5) that are used. For example, the following portion of a composition expression describes the creation of a client node C instantiated from a template τ and its connection to one of the server nodes S_1 and S_2 using a label L with the semantic abbreviation **bac** (bidirectional asynchronous communication):

$$create\_node(C,\tau) \rightarrow assign\_label(L, S_1, C, \mathbf{bac}) \sqcap assign\_label(L, S_2, C, \mathbf{bac})$$

2.1.3 Unit Type

The *Setting* schema and its associated operations show how a module can be constructed from library components. Such a module is a self-contained computational unit without any external connections. The *Unit* schema corresponds to a *unit type* that describes these connections and the associated interface semantics. It includes a set of *virtual ports* that represent the "public" interfaces of the unit and a mapping that gives the attributes of these ports. The schema also includes a *connect* mapping that describes the links between slots and virtual ports and a *virtual_port_descr* mapping that describes the semantics of the virtual ports. This information represents the *interface semantics* of a unit. The *Unit* schema imposes only a minimal restriction on the interface that requires the direction of data movement to be consistent. Further restrictions are derived from application-dependent information about the desired behavior of units. For example, type consistency requirements may be placed on the *connect* mapping, and the *virtual_port_descr* mapping may specify broadcasting or multiplexing behavior for a virtual port. The following example illustrates how unit types can be used to describe an aspect of a hybrid system that includes hardware as well as software.

(c) A computer in a network can be represented as a unit type, where the nodes correspond to sockets, and their connections to virtual ports correspond to the assignment of sockets to services such as ftp or telnet. The *interface_attr* mapping specifies standard information such as the Internet address and port number. Since a port can be linked to more than one socket, ports must support multiplexing of messages. The semantics of a virtual port p is specified as $virtual\_port\_descr(p) = Receive \,\Box\, Send$, where

$$Receive = {}^{*}(\Box p_k?msg \rightarrow socket\_slot_{number(msg)}!msg \rightarrow Skip : k \in K);$$

$number(msg)$ is the local socket number identifying the destination of the message msg, and K corresponds to the computer's set of network connections.

In [RS94], a top-down component-based design methodology was modeled by using a family of generic connector templates. *ASDL* supports a more straightforward representation of this methodology, since it provides an operation to incorporate a special unit type into an existing module. The included unit corresponds to an empty module. Although this unit is not a node, it has slots that can be used to specify connections to other nodes. The internal structure of such a unit can be constructed later, thereby providing the desired top-down design capability.

2.1.4 System Type

The *System* schema includes the *Library* schema as well as architectural state information: the modules and units that have been used to describe logical views of a component-based system, the relationships between units and modules, the connection between the library templates that have been constructed by encapsulating units and the corresponding unit types, and architectural connections between different logical views.

2.1.5 Operations

ASDL provides a number of generic operations that support the incremental specification of the software types by updating the *System* schema. These serve as guides for the design of application-dependent operations that are constructed by adding new signatures and constraints to existing operations, or by incorporating existing operations into a new operation. The operations include *setting operations* to create and delete nodes, assign labels to slots, specify a composition expression, and select semantic abbreviations, *interface operations* to specify virtual ports, attributes, links, and virtual port descriptions, an *encapsulation operation* to create a new library template from a unit, and *relation operations* to specify and modify relations between units.

The encapsulation operation creates a new library template from an existing unit type. The virtual ports of the unit type become the ports of the template and the attributes of these ports are derived from the unit's interface. The template's interpretation is derived from the interpretations of the templates underlying the nodes, the composition expression, the abbreviations of labels, and the semantics of the virtual ports. This represents a complex synthesis of the semantics of the entities associated with the unit. In general, it may not be possible to construct a "closed form" interpretation of a template, due to the presence of partial and heterogeneous information. The new library template can, in turn, be used to create a node in another module. *ASDL* permits an application-dependent interpretation of the extent to which the internal structure of the node is visible in the new module. For example, if encapsulation requires that each virtual port is linked to a node in the underlying unit, then one interpretation is that only the resources of the nodes linked to the port can be accessed through the port. On the other hand, if encapsulation permits a virtual port with no links, then another interpretation may allow characteristics of the node to be modified by using the port.

3 Comparison

The perspective described here is most similar to that of Dellarocas [Del96], who proposes a language for classifying and describing typical features of the interconnections between software components. Dellarocas has also developed tools that implement these features and use them to integrate components into a system. The *Component-Based Software Engineering* project at Andersen Consulting [BBK+97] takes a similar approach. Since the methodologies associated with these approaches only provide relatively informal descriptions of component interconnections, they do not seem to have the power and flexibility of *ASDL*.

Alexander and his associates [BA96, PA97b, PA97a] have proposed a specification-oriented approach to component-based software engineering. The work in [BA96] is particularly relevant to the perspective described here; it uses a combination of *Larch* [GH93] and *CSP* that is analogous to the *Z/CSP* formalism of *ASDL*. The formalism of [BA96] is used to develop problem-specific applications for software, hardware, and hybrid systems. While *ASDL* (and *Z*) seem more appropriate for modeling domain-specific component-based software development systems, there are interesting similarities between the hardware models of [BA96] and the model in example (c) above, and also between the way in which nonfunctional requirements are represented by the constrained *by* clause of *VSPEC* [BA96] and the semantic abbreviations of *ASDL*.

Batory and his associates have used a generative approach [BO92] to describe how software systems can be constructed from plug-compatible components. In this approach, a type equation is used to define a software system, and an attribute grammar is used to express the set of all software systems that can be generated. This approach has been applied to network protocols and database management systems [BO92], and more recently to memory simulators [JPB97] and data language compilers [VB97]. While Batory's approach seems well suited to these applications, it relies on type-checking of parameters to determine the validity of a specific component composition. This relatively rigid approach is less appropriate for systems like *PGM*, in which the validity of component compositions depends on the values of attributes of component interface features.

Several researchers have proposed formalisms for describing software architectures and architectural styles [AAG95, AG94, IW95, MDEK95, ZJ93, Wil96]. While the spirit of all of this work has much in common with our own ideas, we believe that there are some important differences. First, the software types include a comprehensive set of variable declarations and constraints. These provide expressive power and flexibility, and also permit the use of a mixture of formal notations for the specification of architectural features. *ASDL* therefore supports an approach to architectural specifications that is both heterogeneous and orthogonal. Second, the operations provide a natural way to describe architectural styles that have dynamic features. This approach was illustrated in example (b) above, where the special event names used in the composition expression referred to the operations *create_node* and *assign_label*. Some approaches [IW95, MDEK95] use languages that support the dynamic creation of processes. These languages can also be used to model evolving architectures, but they have the drawback of additional semantic complexity when compared with *CSP* and *Z*. In addition, the dynamic operations must be explicitly specified in the underlying languages. In *ASDL*, only a reference to an operation like *create_node* is needed - the actual operation may be specified using a more appropriate formalism.

The increasing scale and complexity of modern software systems requires that they often be assembled from existing components. The goal of our approach is to develop and apply a formalism that can be used to describe methodologies that support the component-based construction of software. The fact that the software types and operations of *ASDL* are described in terms of *Z* schemas is not important per se - the key point is that the language provides a system-independent architectural notation. It is also important to note that *Z* and *CSP* are used orthogonally, in order to express different aspects of *ASDL*.

Finally, the software types and operations provide not only the ability to describe particular component-based software development systems, but also a basis for a methodology for developing architectural descriptions. This is analogous to the manner in which abstract data types are used. A stack with its standard operations suggests potential uses in algorithms; a software type with its associated operations similarly suggests potential uses in architectural descriptions. For example, given a module type representing a logical view, it is natural to formulate interface requirements for a corresponding unit in terms of operations that specify virtual ports, attributes, links, and virtual port descriptions. As another example, in describing an application system, it is natural to ask whether two nodes belong to a common view and should be connected by an *assign_label* operation, or whether they belong to different views and should be related by an operation that creates entries in a relation between the views. The types and operations of *ASDL* provide a rich variety of choices for handling such issues.

References

[AAG95] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[AG94] R. Allan and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, 1994.

[BA96] P. Baraona and P. Alexander. Abstract architecture representation using VSPEC. Technical report, University of Cincinnati, Cincinnati, OH, 1996.

[BBK+97] F. Bronsard, D. Bryan, W. Kozaczynski, E. S. Liongosari, J. Q. Ning, A. Ólafsson, and J. W. Wetterstrand. Towards software plug-and-play. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 136–145, 1997.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1:355–398, 1992.

[Del96] C. Dellarocas. A coordination perspective on software architectures: towards a design handbook for integrating software components. Technical Report 193, Massachusetts Institute of Technology, Cambridge, MA, 1996.

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[IW95] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the Chemical Abstract Machine model. *IEEE Transactions on Software Engineering*, 21:373–386, 1995.

[JPB97] G. Jiménez-Pérez and D. Batory. Memory simulators and software generators. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 19–29, 1997.

[KS95] D. Kaplan and R. Stevens. Processing graph method 2.0 semantics. Technical report, Naval Research Laboratory, Washington, DC, 1995.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, 1995.

[PA97a] J. Penix and P. Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 1997.

[PA97b] J. Penix and P. Alexander. Towards automated component adaptation. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, 1997.

[RS94] M. Rice and S. Seidman. A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, January 1994.

[RS96a] M. Rice and S. Seidman. Describing the PGM architectural style. Technical Report CS-96-121, Colorado State University, Fort Collins, CO, 1996.

[RS96b] M. Rice and S. Seidman. Using Z as a substrate for an architectural style description language. Technical Report CS-96-120, Colorado State University, Fort Collins, CO, 1996.

[Spi89] J. Spivey. *The Z notation, A Reference Manual.* Prentice-Hall, 1989.

[VB97] E. E. Villareal and D. Batory. Rosetta: A generator of data language compilers. In *Proceedings of the 1997 Symposium on Software Reusability*, pages 146–156, 1997.

[Wil96] D. Wile. Semantics for the architecture interchange langage, ACME. In *Proceedings of the Second International Software Architecture Workshop*, pages 28–30, 1996.

[ZJ93] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering*, 2(4):379–411, October 1993.

4 Biography

Michael D. Rice is Associate Professor of Computer Science at Wesleyan University in Middletown, CT. His research is in the areas of formal methods, software architectures, and applications of topological ideas to computer science. Before coming to Wesleyan in 1990, Rice was Associate Professor of Computer Science at George Mason University. He received the Ph.D. in Mathematics from Wesleyan University in 1973.

Stephen B. Seidman is Professor of Computer Science and Chair of the Department of Computer Science at Colorado State University in Fort Collins, CO. His research is in the area of formal models for software architectures and software architectural styles. He is also directing a project to develop an *IEEE* standard for the *PGM* software architectural style. Before coming to Colorado State in 1996, Seidman was Professor of Computer Science and Engineering and Head of the Department of Computer Science and Engineering at Auburn University. He came to Auburn from George Mason University, where he was Professor of Computer Science, in 1990. He received the Ph.D. in Mathematics from the University of Michigan in 1969.