An Introduction to the Aesop System

David Garlan[*]
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Version of 11 July 1995

Abstract

As the design of software architectures emerges as a discipline within software engineering, it becomes increasingly important to support architectural description and analysis with tools and environments. This paper provides a brief introduction to Aesop, a set of tools for developing architectural design environments that exploit architectural styles to guide software architects in producing specific systems.

1 Introduction

A critical aspect of any complex software system is its architecture. At an architectural level of design a system is typically described as a composition of high-level, interacting components. Frequently these descriptions are presented as informal box and line diagrams depicting the gross organizational structure of a system, and they are often described using idiomatic characterizations such as "client-server organization," "layered system," or "blackboard architecture."

Architectural designs are important for at least two reasons. First, an architectural description makes a complex system intellectually tractable by characterizing it at a high level of abstraction. In particular, the architectural design exposes the top level design decisions and permits a designer to reason about satisfaction of system requirements in terms of assignment of functionality to design elements. For example, for a system in which data throughput is a key issue, an appropriate architectural design would allow the software architect to make system-wide estimates based on values of the throughputs for the individual components.

Second, architectural design allows designers to exploit recurring patterns of system organization. Such patterns—or *architectural styles*—ease the design process by providing routine solutions for certain classes of problems, by supporting reuse of underlying implementations, and by permitting specialized analyses. Consider for example, an architectural design that uses a pipe-and-filter

[*]Portions of this overview were drawn from "Exploiting Style in Architectural Design Environments" by David Garlan, Robert Allen, and John Ockerbloom", in *Proceedings of SIGSOFT'94: Foundations of Software Engineering* December 1994. The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9357792; and by a grant from Siemens Corporate Research.

style. When mapped to a Unix implementation the system can take advantage of the rich collection of existing filters and the operating system support for pipe communication. As another example, consider the traditional decomposition of a compiler, which has made it possible for undergraduates to build a non-trivial system in a semester course.

While at present the practice of architectural design is largely ad hoc, the topic is receiving increasing attention from researchers and practitioners in areas such as module interface languages, domain-specific architectures, software reuse, codification of organizational patterns for software, architectural description languages, formal underpinnings for architectural design, and architectural design environments. Collectively these efforts are working to put architectural design on a more solid basis and make principles and techniques of architectural design more widely accessible.

As architectural design emerges as a discipline within software engineering, it becomes increasingly important to support architectural description and analysis with tools and environments. Indeed, already we are beginning to see a proliferation of environments oriented around specific architectural styles. These environments typically provide tools to support particular architectural design paradigms and their associated development methods. Examples include architectures based on dataflow, object-oriented design, blackboard shells, and control systems.

Unfortunately each such environment is built as an independent, hand-crafted effort—and at great cost. While development efforts may exploit emerging software environment infrastructure (persistent object bases, tool integration frameworks, user interface toolkits, etc.), the *architectural* aspects are typically redesigned and reimplemented from scratch for each new style. The cost of such efforts can be quite high. Moreover, once built, each environment typically stands in isolation, supporting a single architectural style tailored to a particular product domain.

Aesop is a system designed to help ameliorate the situation. Aesop provides a toolkit for constructing open, architectural design environments that support architectural styles. The basic idea is that Aesop makes it easy to define new styles and then use those styles to create architectural designs. In brief, each Aesop environment is configured around a set of styles, which guide the software architect in creating a design for a new system. Underlying each design environment, Aesop provides a set of basic support functions for architectural design: a design manager for storing and retrieving designs; a graphical user interface for modifying and creating new designs; a tool integration framework that makes it relatively easy to add new tools (such as compilers, architectural analysis tools, etc.) to the environment; and a repository mechanism for reusing fragments and patterns from previous designs.

In the remainder of this overview we explain in more detail what we mean by architectural style, and outline how Aesop works.

2 What is Architectural Style?

While there is currently no single well-accepted definition of software architecture it is generally recognized that an architectural design of a system is concerned with describing its gross decomposition into computational elements and their interactions [PW92, GS93, GP95]. Issues relevant to this level of design include organization of a system as a composition of components; global control structures; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives.

It is possible to describe the architecture of a particular system as an arbitrary composition of idiosyncratic components. However, good designers tend to reuse a set of established architectural organizations—or *architectural styles*. Architectural styles fall into three broad (overlapping) categories.

Organizational Structures This category includes global organizational structures, such as layered systems, pipe-filter systems, client-server organizations, blackboards, etc.

Patterns: This category includes localized patterns, such as model-view-controller and many other object-oriented patterns [GHJV94]. Unlike organizational idioms, which provide a broader design vocabulary, most patterns tend to focus on a small portion of a system's structure, and provide specialized solutions to specific localizable problems.

Reference models: This category includes system organizations that prescribe specific (often parameterized) configurations of components and interactions for specific application areas. A familiar example is the standard organization of a compiler into lexer, parser, typer, optimizer, code generator [PW92]. Other reference architectures include communication reference models (such as the ISO OSI 7-layer model [McC91]), some user interface frameworks, and a large variety of domain-specific approaches in areas such as avionics [BV93] and mobile robotics [SLF90, HR90].

In terms of their role in architectural design, architectural styles typically determine four kinds of properties:

1. They provide a *vocabulary* of design elements—component and connector types such as pipes, filters, clients, servers, parsers, databases, etc.

2. They define a set of *configuration rules*—or topological constraints—that determine the permitted compositions of those elements. For example, the rules might prohibit cycles in a particular pipe-filter style, specify that a client-server organization must be an n-to-one relationship, or define a specific compositional pattern such as a pipelined decomposition of a compiler.

3. They define a *semantic interpretation*, whereby compositions of design elements, suitably constrained by the configuration rules, have well-defined meanings.

4. They define *analyses* that can be performed on systems built in that style. Examples include schedulability analysis for a style oriented toward real-time processing [Ves94] and deadlock detection for client-server message passing [JC94]. A specific, but important, special case of analysis is code generation: many styles support application generation (e.g., parser generators), or enable the reuse of code for certain shared facilities (e.g., user interface frameworks and support for communication between distributed processes).

The use of architectural styles has a number of significant benefits. First, it promotes design reuse: routine solutions with well-understood properties can be reapplied to new problems with confidence. Second, use of architectural styles can lead to significant code reuse: often the invariant aspects of an architectural style lend themselves to shared implementations. For example, systems described in a pipe-filter style can often reuse Unix operating system primitives to implement task scheduling, synchronization, and communication through pipes. Similarly, a client-server style can take advantage of existing RPC mechanisms and stub generation capability. Third, it is easier for others to understand a system's organization if conventionalized structures are used. For example, even without giving details, characterization of a system as a "client-server" organization immediately conveys a strong image of the kinds of pieces and how they fit together. Fourth, use of standardized styles supports interoperability. Examples include CORBA object-oriented

Need a figure like the one on the aesop home page.

Figure 1: Developing Environments with Aesop

architecture, the OSI protocol stack, and event-based tool integration. Fifth, as we have noted, by constraining the design space, an architectural style often permits specialized, style-specific analyses. For example, it is possible to analyze systems built in a pipe-filter style for schedulability, throughput, latency, and deadlock-freedom. Such analyses might not be meaningful for an arbitrary, ad hoc architecture – or even one constructed in a different style. In particular, some styles make it possible to generate code directly from an architectural description. Sixth, it is usually possible (and desirable) to provide style-specific visualizations. This makes it possible to provide graphical and textual renderings that match engineers' domain-specific intuitions about how their designs should be depicted.

3 What is Aesop?

Given these benefits, it is perhaps not surprising that there has been a proliferation of architectural styles. In many cases styles are simply used as informal conventions. In other cases – often with more mature styles – tools and environments have been produced to ease the developer's task in conforming to a style and in getting the benefits of improved analysis and code reuse.

To take two illustrative industrial examples, the HP Softbench Encapsulator [Fro89] helps developers build applications that conform to a particular Softbench event-based style. Applications are integrated into a system by "wrappping" them with an interface that permits them to interact with other tools via event broadcast. Similarly, the Honeywell MetaH language and supporting development tools provide an architectural description language for real-time, embedded avionics applications [Ves94]. The tools check a system description for schedulability and other properties and generate the "glue" code that handles real-time process dispatching, communication, and resource synchronization.

While environments specialized for specific styles provide powerful support for certain classes of applications, the cost of building these environments can be quite high, since typically each style-oriented tool or environment is built from scratch for each new style. We believe that an effective discipline of software architecture requires a way to more easily develop automated support for defining new styles and incorporating those definitions into environments that can take advantage of them.

Aesop was designed to do just that. Aesop is a system for developing style-specific architectural development environments. Figure 1 illustrates the basic idea behind the system: A set of styles are loaded into Aesop to produce an environment tailored to those styles.

Using the information provided by the style descriptions and some shared infrastructure common to all Aesop environments, each of these environments supports:

1. a palette of design element types (i.e., style-specific components and connectors) corresponding to the vocabulary of the style
2. checks that compositions of design elements satisfy the topological constraints of the style
3. optional semantic specifications of the elements
4. an interface that allows external tools to analyze and manipulate architectural descriptions
5. multiple style-specific visualizations of architectural information together with one or more graphical editors for manipulating them.

PF snapshot goes here.

Figure 2: A Snapshot from an Aesop Environment

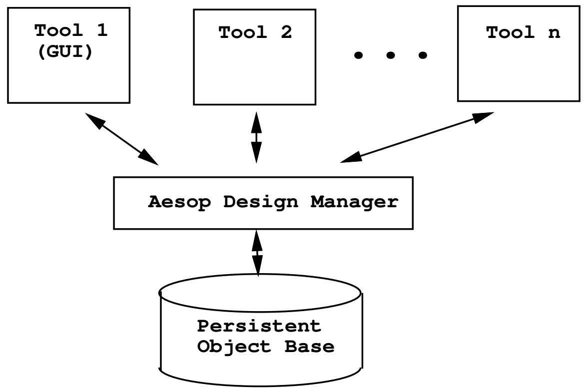

Figure 3: The Structure of an Aesop Environment

6. a design repository for storing and retrieving designs
7. one or more architectural reuse libraries for storing and retrieving existing design fragments

To illustrate, consider the snapshot in Figure 2. This picture shows an Aesop software architecture design environment that has been loaded with the "Unix-Pipe-and-Filter" style. The strip of buttons on the right side of the picture allows the user to select from a palette of primitive design elements in this style. The boxes in the design represent components of the system (filters in this case), and the lines with circles in the middle represent the connectors (pipes).

4 How Does Aesop Work?

Aesop adopts a conventional structure for its environments: each environment is organized as a collection of tools that share data through a persistent object base. (See Figure 3.) The object base runs as a separate server process and provides typical database facilities: transactions, concurrency control, persistence, etc. (In our current version, the database is built on top of the Exodus storage manager from the University of Wisconsin.)

Tools run as separate processes and access the object base through an RPC interface called the "Aesop Design Manager", which defines operations for creating and manipulating architectural objects. This interface is defined as a set of object types that are linked with tools that intend to directly manipulate architectural data. Additionally, tools can register an interest in specific data objects, and will be notified when they change. This same mechanism also serves to integrate external tools. For example, in the pipe-filter environment, described above, code is generated by announcing a message to a suitably "encapsulated" code generation tool. Tools such as external editors are handled in the same way.

The user interface to an Aesop environment is centered around a graphical editor and database browser provided by the Aesop system. This tool can be customized to provide style-specific

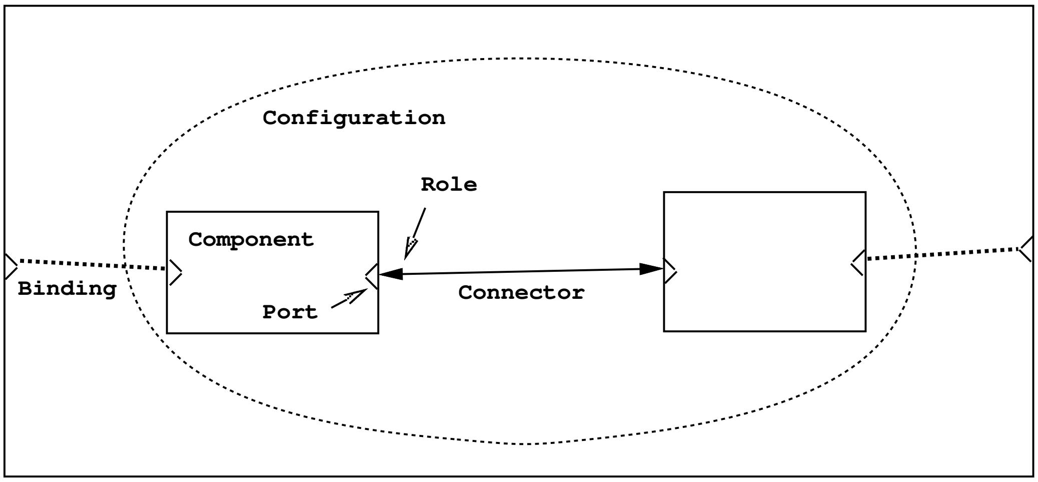

Figure 4: Generic Elements of Architectural Description

displays and views. The current graphical editor is based on Tcl/Tk [Ous94]. While this editor is provided as a default, it is important to note that it runs as a separate tool, and could be replaced or augmented with other interface tools.

Given a persistent object base for architectural representation, an important question is what are the types of objects that can be stored in the database.

Our approach to architectural representation is based on a generic ontology of seven entities: components, connectors, configurations, ports, roles, representations, and bindings. (See Figure 4.)

The basic elements of architectural description are *components*, *connectors* and *configurations*. Components represent the loci of computation; connectors represent interactions between components; and configurations define topologies of components and connectors. Both components and configurations have interfaces. A component interface is defined by a set of *ports*, which determine the component's points of interaction with its environment. Connector interfaces are defined as a set of *roles*, which identify the participants of the interaction.

Because architectural descriptions can be hierarchical, there must be a way to describe the "contents" of a component or connector. We refer to such a description as a *representation*.

For such descriptions there must also be a way to define the correspondence between elements of the internal configuration and the external interface of the component or connector. A *binding* defines this correspondence: each binding identifies an internal port with an external port (or, for connectors, an internal role with an external role).[1]

In the Aesop system this ontology is realized as fixed set of abstract class definitions: each of the seven types of architectural building block is represented as a class. Operations supported by these classes include adding and removing ports to components, connecting a connector role to a component port, establishing a binding between two ports or two roles, adding a new representation to a component or connector, etc.

In many cases, representation of a component or connector is not architectural, per se. For example, a component might have a representation that specifies its functionality, or a code module that describes an implementation. Similarly, a connector might have a representation that specifies

[1] Note that bindings are not connectors: connectors define paths of interaction, while bindings identify equivalences between two interface points. Moreover, connectors always associate a roles with a port, while a binding associates a port with another port, or a role with another role.

its protocol [AG94]. That information is often best manipulated by external non-architectural tools, such as compilers and proof checkers, and stored in an external database (such as the file system). To accommodate such external data, we provide a subtype of representation called *external_rep*, which in turn has other subtypes such as *text_file_rep*, *oracle_rep*, *ast_rep*. These references are usually interpreted by the tools that access them. External representations thus provide external data integration for Aesop environments.

5 How are Styles Defined?

The generic object model provides the foundation for representing architecture. However, to obtain a useful environment, that framework must be augmented to support richer notions of architectural design. In Aesop this is done by specifying a style.

The model that we have adopted for style definition is based on the principle of subtyping: a style-specific vocabulary of design elements is introduced by providing subtypes of the basic architectural classes or one of their subtypes. Stylistic constraints are then supported by the methods of these types.[2] Additionally, a style can identify a collection of external tools: some of these may be specifically written to perform architectural analyses, while others are links to external software development tools.

To see how this scheme is used, let us look at two of the styles supported by the current Aesop release. For each style we (a) outline the design vocabulary, (b) characterize the nature of the configuration rules, (c) explain how semantics are encoded, and (d) describe the analyses carried out by tools in the environment.

5.0.1 A Pipe-Filter Style

A pipe-filter style supports system organization based on asynchronous computations connected by dataflow. In terms

Vocabulary. Figure 5 illustrates the type hierarchy we used to define a pipe-filter style. *Filter* is a subtype of component and *pipe* a subtype of connector. Further, ports are now differentiated into *input* and *output* ports, while roles are separated into *sources* and *sinks*.

Configuration rules. The pipe-filter style constrains the kinds of children and connections allowed in a system. Besides the constraints on port addition described above, pipes must take data from ports capable of writing data, and deliver it to ports capable of reading it. Hence, source roles can only attach to input ports, and sink roles can only attach to output ports.

Semantic interpretation. In the prototype pipe-filter the semantics of filters is given by a simple, style-specific filter language. The associated tool provides typechecking and other static analyses. The semantics of pipes is described formally (but off-line) as in [AG94].

Analyses. In addition to the static semantic checks just outlined, we incorporated a tool for generating code from filter descriptions. Hence, a pipe-filter description can be used to generate a running program, with the help of some style-specific tool and an external editor.

5.0.2 A Real-Time Style

An important class of system organization divides computations into tasks communicating by synchronous and asynchronous messages. Within this general category are systems that must satisfy

[2] We are currently building a "style developer's tool", which will allow many aspects of a style to be specified without any direct coding of methods.

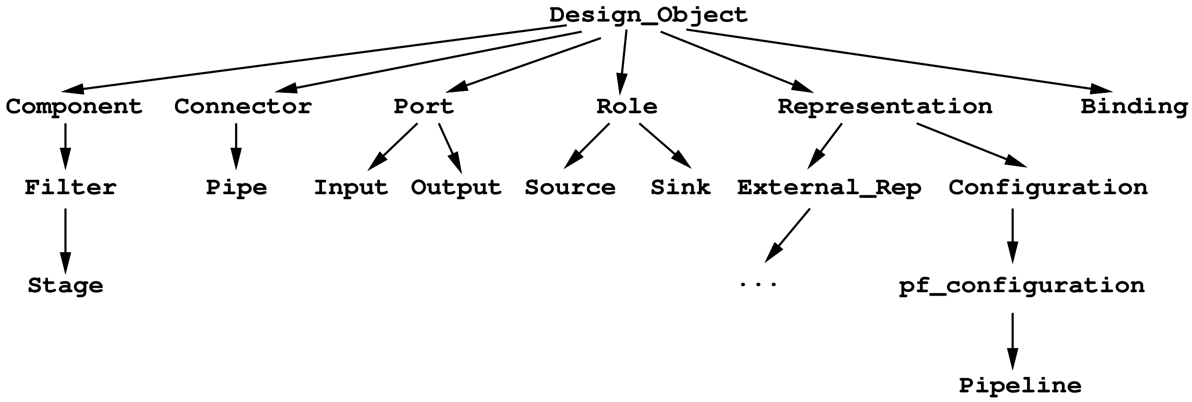

Figure 5: Style Definition as Subtyping

real-time scheduling constraints while processing their data. We created an Aesop environment for an architectural style, developed at the University of North Carolina, that supports the design of such systems [Jef93].

Underlying the architectural style is a body of theory for analyzing real-time systems. This theory allows one to determine the (scheduling) feasibility of a system from the processing rates of its component tasks, rates of inputs from external devices, and shared resource loads. The theory also leads to heuristics for improving the schedulability of a system that is not feasible. The style has been applied primarily to real-time, multi-media applications.

Vocabulary. The real-time style defines three subtypes of component: *devices*, which identify inputs to the system, *processes*, which compute over that data, and *resources*, which support shared resources such as disks, monitors, etc. Components have associated style-specific information about rates of processing and computation loads. There are two new connector types, representing synchronous and asynchronous message passing.

Configuration rules. Configuration rules include: paths through the processing graph must originate with devices; there must be no dangling ports or connectors; communication with resources must be synchronous; and input devices may not have input ports.

Semantic interpretation. The semantic interpretation of a system is determined by the underlying semantics for the connectors, plus the code defined for the tasks. The task code is written in a stylized form, which, like the pipe-filter style, provides syntactic guidance for reading and writing messages to ports. Our system checks that the types of information are consistent across the connectors, but code generation is supported by tools outside our system.

Analyses. The new style enables two kinds of analyses. First, it is possible to detect whether there are resource conflicts. These conflicts arise when multiple processes try to access the same resource in such a way that one or more of the processes will not be able to maintain its processing rate. The second is an analysis of the scheduling feasibility of the system. This determines whether a single CPU can support the specific configuration of devices, processes, and resources. In addition to these analyses, a set of "repair heuristics" are incorporated in a tool that advises the user about possible ways to improve schedulability and resource usage. These heuristics center around decreasing load by cost of shared resources and/or reducing the rates of certain processes. Finally, a style-specific tool allows us to translate our architectural description into one that is readable by external tools built outside our project for code generation and analysis.

5.1 Other Styles

In addition to the two styles just outlined, the demo version of Aesop supports the following styles.

- **Generic Style:** This style includes only the generic architectural vocabulary (components, connectors, etc.). It provides the weakest, but most general form of architectural support.
- **Unix-Pipe-Filter Style:** This style supports the description of simple pipe-filter systems specifically for Unix. It augments the simple Pipe-Filter Style with component types represented by Unix binaries and scripts.

6 How Can I Find Out More?

About Aesop

For an expanded version of this overview the paper [GAO94] contains a more in-depth discussion of the first version of the Aesop System. (Currently we are distributing the second version, and so a number of details have changed.)

Aesop was developed as part of the ABLE Project, whose WWW home page is

`URL: http://www.cs.cmu.edu/Web/Groups/able/`

On-line information on Aesop is also available directly through

`URL: http://www.cs.cmu.edu/Web/Groups/able/aesop/`

Getting a Demonstration Copy of Aesop

Aesop currently runs on Sun workstations under SunOS. It available for release as a demonstration system. To obtain a copy send mail to `aesop-help@cs.cmu.edu`.

Further Reading

In this brief overview we have only scratched the surface of topics such as software architecture, architectural style, and architectural design environments. There is a growing body of literature on each of these topics. As a starting point:

Software Architecture: Two good introductions to software architecture and some common architectural styles are [GS93] and [PW92]. An introductory book on software architecture will be appearing soon [SG95]. A number of architectural description languages have been proposed; each of these elaborates a view of what it means to define software architectures [SDK+95, LAK+95, AG94]. To get a feeling for what is going on in the area, the *Proceedings of the First International Workshop on Software Architecture* is now available. For other examples of current research in software architecture you might look at the *IEEE Transactions on Software Engineering, Special Issue on Software Architecture* April, 1995.

Architectural Style: There have been several attempts to understand and explicate the nature of style. Several of these have been formal [AAG93, MQR95]. Others have examined the general nature of style [Gar95] and ways to represent it [DC95]. As mentioned earlier, the paper [GAO94] has more details on Aesop itself. A number of "domain-specific" style have been investigated. One rich source of material on this topic is [Tra94].

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[BV93] Pam Binns and Steve Vestal. Formal real-time architecture specification and analysis. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[DC95] Thomas R. Dean and James R. Cordy. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4), April 1995.

[Fro89] Brian Fromme. HP Encapsulator: Bridging the generation gap. Technical Report SESD-89-26, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

[Gar95] David Garlan. What is style? In *Proceedings of the First International Workshop on Architectures for Software Systems*, April 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[GP95] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

[HR90] Barbara Hayes-Roth. Architectural foundations for real-time performance in intelligent agents. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, 2:99–125, January 1990.

[JC94] G.R. Ribeiro Justo and P.R. Freire Cunha. Deadlock-free configuration programming. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.

[Jef93] Kevin Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 796–804, Indianapolis, IN, February 1993. ACM Press.

[LAK+95] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[McC91] Gary R. McClain, editor. *Open Systems Interconnection Handbook*. Intertext Publications McGraw-Hill Book Company, New York, NY, 1991.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

[SG95] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1995.

[SLF90] Reid Simmons, Long-Ji Lin, and Christopher Fedor. Autonomous task control for mobile robots. In *Proceedings of the 5th IEEE International Symposium on Intelligent Control*, Philadelphia, PA, September 1990.

[Tra94] Will Tracz. Collected overview reports from the DSSA project. Loral Federal Systems - Owego, October 1994.

[Ves94] Steve Vestal. Mode changes in real-time architecture description language. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 1994.