Architectural Mismatch
or
Why it's hard to build systems out of existing parts*

David Garlan Robert Allen John Ockerbloom

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract

Many would argue that future breakthroughs in software productivity will depend on our ability to combine existing pieces of software to produce new applications. An important step towards this goal is the development of new techniques to detect and cope with mismatches in the assembled parts. Some problems of composition are due to low-level issues of interoperability, such as mismatches in programming languages or database schemas. However, in this paper we highlight a different, and in many ways more pervasive, class of problem: *architectural mismatch*. Specifically, we use our experience in building a family of software design environments from existing parts to illustrate a variety of types of mismatch that center around the assumptions a reusable part makes about the structure of the application in which is to appear. Based on this experience we show how an architectural view of the mismatch problem exposes some fundamental, thorny problems for software composition and suggests possible research avenues needed to solve them.

1 Introduction

Many would argue that future breakthroughs in software productivity will depend on our ability to combine existing pieces of software to produce new applications [BS92]. By constructing new systems out of reusable building blocks it should be possible to create large, high-quality software applications much more rapidly than we now do with the build-from-scratch techniques that dominate most software production.

Over the past decade the broad-based interest in supporting compositional approaches to software has led to considerable investment in research and development in reuse [BP89, Kru92], industry standards for component interaction (e.g., [Cor91]), domain-specific architectures (e.g., [MG92]), toolkits (e.g., [SG86]), and many other related areas.

*This research was sponsored by the National Science Foundation under Grant Number CCR-9357792, by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330, and by Siemens Corporate Research. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the U.S. Government, or Siemens Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

To appear in *Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle WA, April 1995.

However, despite all of this effort systematic construction of large-scale software applications from existing parts remains an elusive goal. Why is this? Clearly some blame can be attributed to the lack of existing pieces to build on, or our inability to locate the desired pieces when they *do* exist. Over time, we may expect progress in this area through the creation of more and better component libraries as well as improved mechanisms to access their contents.

But even when the components are in hand, there remain other fundamental problems that arise because the chosen parts do not fit together well. In many cases mismatches may be caused by low-level problems of interoperability, such as incompatibilities in programming languages, operating platforms, or database schemas. These are hard problems to overcome, but recent research has been making good progress in addressing many of them.

In this experience report we highlight a different, and in many ways more pervasive, class of problem: *architectural mismatch*. Specifically, we use our experience in building a family of software design environments from existing parts to illustrate a variety of types of mismatch that center around the assumptions a reusable part makes about the structure of the application in which it is to appear. Based on this experience we show how an architectural view of the mismatch problem exposes some fundamental, thorny problems for software composition and suggests possible research avenues needed to solve them.

In the next section we begin with a brief, motivating description of the system that we developed. We then continue in section 2.2 by describing the approach that we took in attempting to reuse existing tools and components in the construction process. Section 3 describes the problems that we encountered. In section 4 we analyse the underlying causes of those difficulties. In section 5 we conclude by looking ahead to potential solutions.

2 Wishful Thinking; Harsh Reality

In this section we briefly describe the Aesop system and our development approach. A complete description of Aesop is beyond the scope of this paper. Here we describe just enough to motivate the problems of system composition that we encountered. See [GAO94] for more details on Aesop itself.

2.1 Aesop

For the past five years, the ABLE Project at Carnegie Mellon University has been carrying out research aimed at developing a sound engineering discipline of software architecture. One component of this research is the construction of tools and environments to support architectural design and analysis. The primary thrust of that

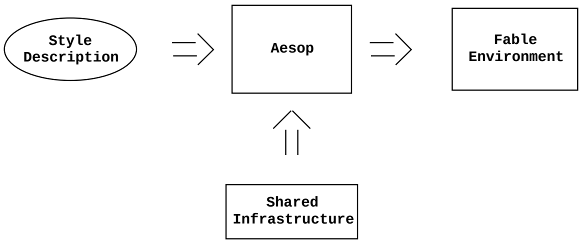

Figure 1: Generating Fables with Aesop

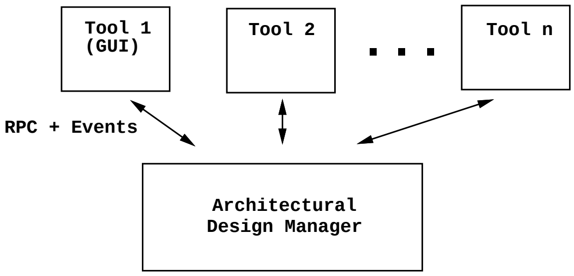

Figure 2: The Structure of an Aesop Environment

development effort has been finding effective mechanisms to exploit *architectural style* [AAG93, PW92]. An architectural style is a recurring pattern of system organization that provides an abstract framework for some family of applications. Architectural styles can be exploited, for example, by simplified analysis of critical system properties [JC94], by concise, understandable system design notations [MG92], and by tools to aid in the design and implementation of complete systems [GAO94].

The Aesop System [GAO94] is our implementation platform for experimenting with style-oriented architectural development environments. Aesop is, in fact, a kind of environment generator: given a description of an architectural style, Aesop produces an environment tailored to that style. As illustrated in Figure 1, Aesop combines a style definition with some shared infrastructure to produce the target environment. The style definition consists of a description of (a) the architectural design vocabulary (as a set of object types), (b) visualizations of design elements suitable for manipulation by a graphical editor, and (c) a set of architectural analysis tools to be integrated into the environment.

Each Aesop environment—the output of the environment construction process—is organized in a similar fashion. As sketched in Figure 2, an Aesop environment is structured as an open collection of tools that access a central object-oriented database. (These tools may also access databases outside of Aesop, but we won't discuss this aspect of the system here.) The database stores architectural designs and provides a high-level, object-oriented interface to the tools. The database is responsible for providing transactional access to shared data, and for enforcing the architectural design constraints specified by a particular style.

The tools run as separate processes and access the database through an RPC mechanism that allows them to make method calls to the database. Additionally, the system includes a tool integration mechanism based on event broadcast [Rei90]. This allows tools to register for changes to database objects and to announce significant events to other tools. Typical tools include a graphical editor for creating and browsing architectural designs, and other style-specific tools for carrying out architectural analyses, such as architectural consistency checkers [AG94], code generators, architectural component repositories, *etc*.

The first prototype of the Aesop system was completed in August 1993. A second prototype has recently been constructed. While we are still in the experimental stages of learning how to exploit architectural style, Aesop has been used to construct a number of distinct style-oriented environments, including environments for several styles of pipe-filter system, one for real-time systems definition, and one for generic architectural construction.

2.2 Building Aesop from Existing Components

Two important challenges in building Aesop were (a) the design of notations and mechanisms to support style definition, and (b) the creation of the environment support infrastructure that would be reused to create each of the target environments. Here we focus on the latter.

Viewed abstractly, the infrastructure required by Aesop environments is hardly novel. Indeed, it is now commonplace to construct environments in this fashion, as open, loosely-integrated collections of tools accessing shared data [T^+88, Bam90]. Moreover, graphical editors are common components of drawing packages, case tools and other user interfaces.

We were optimistic, therefore, that it would be possible to obtain most of the infrastructure needed for Aesop by building on existing software. Specifically we wanted to reuse four standard pieces of software infrastructure:

- an object-oriented database
- a toolkit for constructing graphical user interfaces
- an event-based tool integration mechanism
- a RPC mechanism

There were numerous candidates for each of these classes of software. In making our selections we picked systems that *a priori* seemed to have promise for working well together and within our development environment. These were:

- an object-oriented database: OBST, a public domain OODB.
- a toolkit for constructing graphical user interfaces: InterViews, a UI toolkit, together with Unidraw, a reusable framework for creating drawing editors [LVC89, VL90].
- an event-based tool integration mechanism: Softbench, a commercial event-broadcast mechanism [Ger89].
- a RPC mechanism: Mach RPC Interface Generator (MIG), an RPC stub generator that was well targeted to our host operating system [Dra87].

All we had to do was put the subsystems together, a task considerably simplified by the fact they were all written in either C++ or C, had all been used in many projects, and we had source code for all of the parts.

A piece of cake? Unfortunately, no. Two years later, after considerable effort (approximately 5 person-years), we managed to get the pieces working together in our first Aesop prototype. But even then, the size of the system was huge (although we contributed a relatively small proportion of our own code), the performance was sluggish, and many parts of the system were difficult to maintain without detailed, low-level understanding of the implementations.

What went wrong? One might argue that we were simply poor systems builders, but we suspect that this experience is not unfamiliar to anyone who has tried to compose similar kinds of software. In the remainder of this paper we examine these problems in more detail and then consider some of the root causes for them. Our basic conclusion is that many of the hardest problems are best understood as architectural mismatch problems. Each component makes assumptions about the structure of the environment in which it is to operate. Most if not all of these assumptions are implicit, and many of them are in conflict with each other.

3 Problems Encountered

Focusing on the most salient problems, we encountered six main difficulties with integrating the four existing software subsystems into a new coherent system: excessive code size, poor performance, the need to significantly modify the reused subsystems just to get them to work together, the need to re-invent existing functionality in order to match our intended use, unnecessary complexity of applications built on top of the reused systems, and a complex, error-prone system construction process.

Excessive code size: Our user interface code image alone was over 3 MB, after stripping. Our database server's code image was 2.3 MB after stripping. Even small tools (of, say, 20 lines of code) interacting with our system were over 600K after stripping. In an operating system without shared libraries, running the central components plus the supporting tools (such as external structure editors, specification checkers, and compilers) overwhelmed the resources of a moderate-sized workstation.

Poor performance: The system operated much more slowly than we wished. Some of the problems occurred through tool-database communication overhead. For example, to save the state of a simple architectural diagram (containing, say, 20 design objects) took several minutes when we first tried it out. Even with performance tuning, it still took many seconds to perform such an operation.

Furthermore, the excessive code size also contributed to the performance problem. Under AFS, files are cached at the local workstation *in total* when they are opened. When tools are large, the start-up overhead is also large. For example, start-up time of an Aesop environment containing a minimal configuration of tools took approximately three minutes.

Need to modify external packages: Despite the fact that the reused packages seemed to run "out of the box" in our initial tests, we discovered that once we combined them in a complete system they required significant modifications to make them work together at all. For example, we ended up having to significantly modify the Softbench client event loop (a critical piece of the functionality) for it to work with the InterViews event mechanism. We also had to reverse engineer the memory allocation routines for the OBST database in order to communicate object handles to external tools.

Need to re-invent existing functions: In some cases, modifying the packages was not enough: in addition we had to augment the packages with different versions of the functions that they already supplied. For example, (as detailed later) we were forced to bypass InterViews' support for hierarchical data structures because it did not allow direct, external access to hierarchically nested sub-visualizations. Similarly, we ended up rebuilding a separate transaction mechanism on top of a serverized version of the OBST database software, even thought the original supported transactions. This was because it did not allow us to share transactions across multiple address spaces.

Unnecessarily complicated tools: Many of the architectural tools that we wished to develop on top of the infrastructure were logically simple sequential programs. However, in many cases it was difficult to build them as such. This was because the standard interface to their environment required them to handle multiple, independent threads of computation simultaneously.

Error-prone construction process: As construction of the system progressed, it became increasingly costly to modify the system, both because of the time it took to recompile it, and because seemingly simple modifications (such as the introduction of a new procedure call) would break the automated build routines. The recompilation time was in part due to the code size. But more significantly, it was also because of interlocking code dependencies that required minor changes to propagate (in the form of required recompilations) throughout a majority of the system.

4 Understanding the Causes of the Problems

The creators of the reused subsystems that we imported were neither lazy, stupid, nor malicious. Nor were we using the pieces in ways inappropriate to their advertised scope of applicability. Therefore the root causes must lie at a deeper systemic level. Each of the packages that we used to construct our system made assumptions about the structure of the system and, in particular, the nature of the environment in which they were to operate. Virtually all of our serious problems can be traced back to places where these assumptions were in conflict.

To expose the nature of these mismatched assumptions, it is helpful to view the problem from an architectural perspective. Through that lens a system is viewed abstractly as a configuration of components and connectors [GS93, PW92]. The *components* are the primary computational and storage entities of the system: tools, databases, filters, servers, etc. The *connectors* determine the interactions between the components: client-server protocols, pipes, RPC links, *etc*. These abstractions are typically expressed informally as box and line drawings, although recently formal notations for architectural description have begun to emerge [LAK+95, SDK+95, AG94, IW95].

Figure 2 can be seen as an abstract, architectural rendering of an Aesop-generated environment. At this level of abstraction the main components are a collection of tools and the architectural design manager (which consists primarily of a persistent object base). The main connectors are the RPC and event-broadcast mechanism communication links. The parts that we attempted to import provided an implementation basis for two components — the database (via OBST) and a graphical user interface tool (via InterViews) — and two kinds of connectors — RPC (via MIG) and event-broadcast (via Softbench).

Turning now to the problem of system integration, we can identify at least four main categories of *architectural mismatch* that provide a taxonomic framework for understanding how conflicting assumptions arise.

1. **Assumptions about the nature of the components:** Within this category there are three main areas: (1) *infrastructure*—assumptions about the substrate on which the component is built; (2) *control model*—assumptions about which component(s) (if any) control overall the sequencing of computations; (3) *data model*—assumptions about the way the environment will manipulate data managed by a component.

2. **Assumptions about the nature of the connectors:** Within this category we have two areas: (1) *protocols*—assumptions about the patterns of interaction characterized by a connector; (2) *data model*—assumptions about the kind of data that is communicated.

3. **Assumptions about the global architectural structure:** These include assumptions about the topology of the sys-

tem communications and about the presence or absence of particular components and connectors.

4. **Assumptions about the construction process:** In many cases the components and connectors are produced by instantiating a generic building block. For example, a database is instantiated, in part, by providing a schema; an event broadcast mechanism is instantiated, in part, by providing a set of events and registrations. In such cases the building blocks frequently make assumptions about the order in which pieces are instantiated and combined in an overall system.

In the remainder of this section we show how our system exhibited examples of mismatch in each of these categories.

4.1 Assumptions about the Nature of Components

Infrastructure

One kind of assumption that the packages made about components was the nature of the underlying support that they needed to perform their operations. This support takes the form of additional *infrastructure* that is either provided by the packages or that is assumed to exist and therefore used by the packages.

Each of the packages that we used assumed that they were responsible for providing considerable infrastructure. Since we did not need much of this infrastructure, this contributed to our excessive code size. For example, OBST provided an extensive library of standard object classes to make general purpose programming easier. However, we only needed a small number of these classes, since we have a constrained, special-purpose, data model. (See [GAO94] for a detailed description of that model.)

Additionally, some of the packages made assumptions about the kind of components that would be exist in the final system, and therefore used infrastructure that did not match our needs. For example, the Softbench Broadcast Message Server expected all of the components to have a graphical user interface and therefore used the X Server to provide communication primitives. This meant that even those tools that did not have their own user interface (such as compilers or automatic design manipulation utilities) were required to include the entire X library in their executables.

Control Model

One of the most serious problems that we encountered is the result of the assumptions made by the packages about what part of the software held the main thread of control. Three of the packages, Softbench, InterViews, and the Mach Interface Generator, used an event loop to deal with communication events. The event loop encapsulates the details of the communication substrate and allows the developer to modularize the components' structure as callbacks.

Unfortunately, in each of the three packages they used a different event loop. Softbench based its main thread of control on the X Intrinsics package. InterViews provided its own, object-based abstraction of an event loop implemented directly in terms of Xlib routines. MIG had a handcrafted loop for the server to wait for Mach messages. In each case the control loop was provided as part of the package, and in each case the control loop was not compatible with the others. [1]

This meant that we had to reverse engineer the InterViews event loop and modify it to poll for Softbench events before we were able to have the user interface respond to events. In the time that we had available for the project, we were unable to modify the MIG control loop so that the server could receive events, although we had originally seen this as an important way of providing modular control over the design data.

Data Model

Another assumption made by the packages about components was the nature of the data that they would be manipulating. For example, Unidraw provided a hierarchical model for its visualization objects. One object could be part of another object, and any manipulation of the parent (such as changing its position on the screen) would result in a corresponding change in the child. The critical assumption made by Unidraw, however, was that *all* manipulations would be of top-level objects. Thus, it was not possible to change a child object except by having the parent manipulate it. While the data that we wanted to present and manipulate was strongly hierarchical, it was important that the user have direct control over the child objects as well as the root objects. Thus we were faced with two alternatives: either to modify Unidraw to support the direct manipulation of children, or to create a flat Unidraw data structure and to build our own, parallel, hierarchy to support the correspondences that we wanted. For our purposes, it turned out to be less costly to re-implement the hierarchy from scratch.

4.2 Assumptions about the Nature of the Connectors

Protocols

When we began the project, we expected to have two kinds of interactions between tools. The first interaction, a pure event broadcast, involved one tool informing others about the state of the world. For example, the database would announce notifications that a particular data object had changed. The second interaction, a request/reply pair, would provide a simple means for multiple tools to be combined to perform a complex manipulation. This connector follows the model of a procedure call in that the requesting tool cannot generally continue usefully until the recipient of the message has completed its task.

The Softbench Broadcast Message Server provides both these kinds of interaction by having different kinds of messages. The first kind of interaction is handled by a "Notify" message, which is announced and then forgotten (by the announcing tool). The second interaction is handled by a pair of event kinds, "Request" and "Reply" messages.

The problem that we encountered arises because the BMS attempts to handle both of the interaction kinds uniformly. In order to receive any message, a tool registers a "callback" procedure for that message. When the message arrives, the Softbench client library invokes the callback procedure. This callback technique is used for all three kinds of message. This means that when a tool makes a request, it does not simply make the request, wait for the reply, and then continue—as one would expect. Instead, the tool must divide its manipulation into two callback routines, one to be done before the call and one to be done after the reply is received. This breaks up the natural structure of the tool and makes it difficult to understand.

More importantly, if any other messages (such as change notifications) are received by the BMS before the reply message, then those will be delivered to the tool and their callbacks invoked before the reply is processed. This means, in effect, that if a tool wishes to delegate any part of its processing, then it must be able to deal with multiple threads of control simultaneously, one for each different notification message that might be delivered before a reply is received. Thus, Softbench's handling of the request/reply protocol forces tools to handle concurrency even if they would be much simpler to construct and understand as sequential programs. This led us to use Mach RPC instead of Softbench for the database

[1] Note that since the event loops were operating in the same process, it was not possible to use simple event gateways to bridge different event-control regimes.

interaction since it is the most critical and heavily used request path in our system.

Data Model

Just as the packages made assumptions about the kind of data that would be manipulated by the components, so did they make assumptions about the data that would be communicated over the connectors. In particular, the two communication mechanisms that we used, Mach RPC and Softbench, made different assumptions about the data. Mach RPC is intended to support integration between arbitrary C programs, and so it provides a C-based model: data passed through procedure calls is based on C structs and arrays. Softbench, on the other hand, assumes that most communication would be about files and the data contained in them, and so all data to be communicated by Softbench is represented as ASCII strings. Since the main kind of data manipulated by our tools was database and C++ object pointers, we were required to develop a translation routines and intermediate interfaces between the different models. This also caused the most significant performance bottlenecks in the system due to translation overheads on every call to the database. The problem occurred despite the fact that we were working in C and C++ exclusively, and that we had compatible data models in all of the components that we developed.

4.3 Assumptions About Global Architectural Structure

It turned out that OBST assumes that the communications in the system formed a star with the database at the center. Specifically, it assumed that all of the tools would be completely independent of each other. This assumption meant that there would be no direct interactions between tools, and so any concurrency among tools represented conflict rather than cooperation. Based on this assumption, OBST selected a per tool, blocking transaction mechanism. Since, as we mentioned earlier, the tools in our environment could coordinate their efforts by delegating part of their computation to other tools, this model was unacceptable; either cooperating tools would deadlock by holding conflicting locks, or conflicting tools could create inconsistencies when a tool attempted to release the database to a cooperating tool.

4.4 Assumptions about the Construction Process

Several of the packages assumed that there were three categories of code being combined in the system: first, existing infrastructure (such as the X libraries and the package's own runtime libraries) that would not change; second, application code developed in a generic programming language that would use the infrastructure but otherwise be self-contained; third, code developed using the notations developed specifically for the package that would control and integrate the rest of the application.

These assumptions (shown graphically in figure 3) led to a particular process for building the system: First the generic parts of the application are built and possibly specified for the package's build tool, and then the package specific sections are preprocessed, compiled, and then linked. Generally, a change to the interface of the generic section required both a respecification and rebuild of the package specific section. This makes sense in isolation, because we think of the packages as providing "glue" code to integrate the parts of the generic application. For example, the Mach Interface Generator assumed that the rest of the code was a flat collection of C procedures, and that its specification described the signature of all of these procedures.

In our case, however, we had multiple packages that made these kinds of assumptions. This meant that there were in fact four categories of code: the three previous categories plus the code generated by other tools. It was the integration of the code generated by different tools that presented us the most difficulty in the build process. This was because we had to take the generated code and make it look like whatever "generic" structure the other tools were expecting. (These dependencies are shown in figure 4.) So, for example, we had to take the output of the OBST pre-processor and specify the resulting procedure calls in the notation of the Mach Interface Generator, run MIG to generate a serverized version of the database, and then rebuild all of the tools (including rebuilding and linking the Softbench wrapper code) to recognize the new client interface. It was these conflicting assumptions about the steps in the construction process that resulted in lengthy and complicated builds.

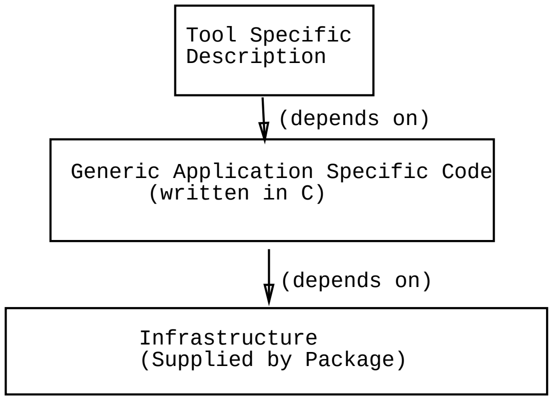

Figure 3: The Assumed Dependency Structure

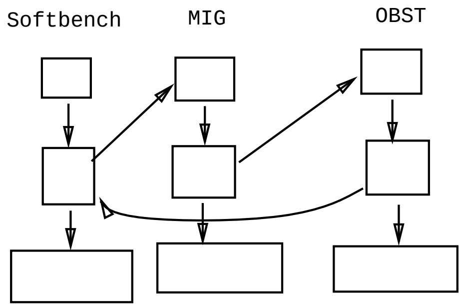

Figure 4: The Actual Dependency Structure

5 The Way Forward?

We believe that the kind of experiences we encountered are common whenever large-scale components are assembled into a new system. And we are convinced that the really hard problems do not go away once you solve low-level issues such as language interoperability, platform independence, and heterogeneous data manipulation. Further, we have suggested that the root causes can best be understood in terms of architectural mismatches.

What can be done about this? There seem to be two complementary approaches that can improve the situation. The first is to improve the way we build components that are intended to be composed into larger systems. The second is to provide new notations, mechanisms, and tools that will enable designers to do a better job of this. In particular, we would single out four necessary aspects of a long-term solution:

1. *Make architectural assumptions explicit.* One of the most significant problems is that architectural assumptions made by a reusable component are never documented. Part of the problem is attributable to a culture in which documentation is generally lacking. But it is also the case that we have lacked the proper vocabulary and structuring techniques for expressing these assumptions. For example, while it may be common for good documentation of an abstract datatype to explicitly list preconditions for calling the interface routines, we have no comparable conventions or theory for documenting many of the architectural assumptions listed in Section 4. While documenting assumptions will not make mismatches disappear, at least it will be possible to detect problems early on. Some initial steps towards this goal can be seen in recent work on architecture description languages [AG94, LAK+95, SDK+95].

2. *Construct large pieces of software using orthogonal sub-components.* While most large reusable subsystems are themselves constructed out of smaller sub-components, it is rare that one can separate the pieces or change the way in which those sub-components are combined. As Parnas has long argued [Par72], each module should hide certain design assumptions. Unfortunately, the architectural design assumptions of most systems are spread throughout the constituent modules. Ideally one would like to be able to tinker with the architectural assumptions of a reused system by substituting different modules for the ones already there. An example of such orthogonalization can be found in recent work aimed at adopting a building-block approach to systems construction, such as [BO92, BSST93].

3. *Provide techniques for bridging mismatches.* Even with good documentation and orthogonal modularization, mismatches will inevitably occur. Currently such mismatches must be dealt with by brute force: hacking low-level communication code, interposing special purpose data translators, etc. We would like to see a much more scientifically based approach to this. At the very least it would be desirable to have a cookbook of standard techniques. Better still would be tools to aid with wrapping, data translation, etc. Recent research addressing this issue includes attempts to automate the use of mediating protocols and wrapper construction [YS94].

4. *Develop sources of architectural design guidance.* Developing good intuitions about what kinds of architectural components work well together is not easy. Currently it is done by trial and error, and it takes skilled designers many years to acquire expertise at putting systems together from parts. Even then this expertise is typically confined to a specific application domain (such as management information systems, or signal processing). We need to find ways codify and disseminate principles and rules for composition of software. Progress towards this goal can be seen in the development of handbooks for reuse of design patterns [GHJV94], architectural design environments [GAO94], and design tools for certain application domains [Lan90].

Acknowledgements

Aesop embodies many ideas from collaborative work with fellow researchers. In particular, our work has been strongly influenced by Daniel Jackson, Mary Shaw, and Jeannette Wing, whom we gratefully acknowledge. We would also like to thank the students and staff who have contributed to the Aesop implementation described in this paper: Mike Baumann, Steven Fink, Doron Gan, Huifen Jiang, Curtis Scott, Brian Solganick, and Peter Su. Finally, we thank Ralph Johnson and John Vlissides for their comments on an earlier draft of this paper.

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.

[Bam90] J. Bamberger. STARS/users workshop: Final report - issues for discussion groups. Technical Report CMU/SEI-90-TR-32 ADA235776, Software Engineering Institute (Carnegie Mellon University), 1990.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[BP89] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability*. ACM Press, 1989.

[BS92] Barry Boehm and William Scherlis. Megaprogramming. In *Proceedings of Software Technology Conference, DARPA*. ARPA, 1992.

[BSST93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of SIGSOFT'93*, pages 191–199, December 1993.

[Cor91] The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[Dra87] R. R. Draves. MIG - the MACH interface generator, August 1987. Comment 1 by schlenk, Sat Jul 2 15:08:47 1988 MIG seem very similiar to the SUN rpcgen facility. Parameters are described in a formal language and compiled into stubs.

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

[Ger89] Colin Gerety. HP Softbench: A new generation of software development tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

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

[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.

[IW95] Paola Inverardi and Alex Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering*, 1995. To appear.

[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.

[Kru92] Charles W. Krueger. Software reuse. *Computing Surveys*, 24(2):131–183, June 1992.

[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*, 1995. To appear.

[Lan90] Thomas G. Lane. A design space and design rules for user interface software architecture. Technical Report CMU/SEI-90-TR-22 ESD-90-TR-223, Carnegie Mellon University, Software Engineering Institute, November 1990.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Compusing user interfaces with interviews. *IEEE Computer*, 22(2), February 1989.

[MG92] Erik Mettala and Marc H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[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.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, July 1990.

[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*, 1995. To appear.

[SG86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[T+88] Richard N. Taylor et al. Foundations for the Arcadia environment architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, MA, November 1988. Published as SIGPLAN NOTICES, 24(2).

[VL90] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[YS94] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. *Proceedings of OOPSLA'94*, October 1994.