Exploiting Style in Architectural Design Environments

David Garlan Robert Allen John Ockerbloom
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

As the design of software architectures emerges as a discipline within software engineering, it will become increasingly important to support architectural description and analysis with tools and environments. In this paper we describe a system for developing architectural design environments that exploit architectural styles to guide software architects in producing specific systems. The primary contributions of this research are: (a) a generic object model for representing architectural designs; (b) the characterization of architectural styles as specializations of this object model; and (c) a toolkit for creating an open architectural design environment from a description of a specific architectural style. We use our experience in implementing these concepts to illustrate how style-oriented architectural design raises new challenges for software support environments.

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. Second, architectural design allows designers to exploit recurring patterns of system organization. As detailed later, 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.

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 an explicit discipline within software engineering, it will become 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 data flow [Mak92], object-oriented design [R^+91], blackboard shells [Nii86], control systems [BV93], and reactive integration [Fro89].

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.

In this paper we describe an approach that helps ameliorate the situation. Focusing on the issue of architectural style we show how to adapt the principles and technology of generic software development environments to provide style-specific architectural support. Specifically, we show how to generate architectural design environments from a description of an architectural style. Like general-purpose environment technology, this approach is not committed to a particular architectural style or development method. But unlike general-purpose approaches, we provide specific mechanisms to define new architectural styles and to use those styles for designing new systems.

In the remainder of this paper we describe a system – called *Aesop* – for developing style-oriented architectural design environments. As we will show, the primary contributions of this research are: (a) a generic object model for representing architectural designs; (b) the characterization of architectural styles as specializations of this object model (through subtyping); and (c) a toolkit for creating an open architectural design environment from a description of a specific architectural style.

Proceedings of the ACM SIGSOFT '94 Symposium on Foundations of Software Engineering, December, 1994, New Orleans, LA.

2 Related Work

2.1 Software Development Environments

For the past decade there has been considerable research and development in the area of automated support for software development: tool integration frameworks [B+88, Ger89], environment generators and toolkits [RT89, vLDD+88, DGHKL84], process-oriented support [KFP88, T+88], etc. These facilities typically provide generic support for some aspects of software development, and can be specialized or instantiated for a particular development environment. Inputs to the specialization process include such things as a BNF description of a programming language, a lifecycle model, a process model, a set of broadcast message definitions, etc.

Our work builds on this heritage (both philosophically and materially), but focuses on the specific task of architectural design. We use the standard building blocks of software development environments to construct style-specific environments: databases, tool integration frameworks, structure-editor generators, user interface frameworks, etc. However, as we describe in Section 4, we have tailored these building blocks to the specific task of describing and analyzing architectural designs.

Consequently, our work complements existing technology for software development support environments, and dovetails nicely with it. In particular, the architectural design environments produced by our system can coexist with existing software development tools and environments.

2.2 Software Architecture

Within the emerging field of software architecture research there are three closely related subareas. The first area is environments that support specific architectural styles. As outlined above, we share with those efforts the goal of supporting architectural development and exploiting architectural styles. However, our work attempts to reduce the cost of building such environments by providing a common basis for implementing them – or at least certain key parts of them. Hence, our research is attacking a more general problem.

The second area is research aimed at providing a rigorous basis for architectural specification and design [GN91, AG92, AAG93, PW92]. To the extent that such research clarifies the nature of architectural representation and the meaning of architectural style, our work builds on those results. In particular, the basic model of architectural representation (Section 4.3) and the elements of style description (Section 3) emerged as a result of our own experience with formalization of architecture. Moreover, tools that have resulted from efforts to formalize software architecture (e.g., architectural compatibility checkers [AG94b] and refinement tools [MQR94]) are natural candidates for tools in our style-specific environments.

The third is research on languages for architectural description. These efforts have focused on providing general-purpose architectural description languages, linguistic mechanisms for component specification and generation, and tools to support these. Within this general area, the two systems that are most closely related to ours are Luckham's Rapide System [LAK+95] and Shaw's UniCon System [SDK+95]. Rapide provides a general-purpose system description language (based on events and event patterns) together with tools for executing and monitoring systems described in the language. UniCon provides a general-purpose architectural description language and a tool that (currently) focuses on the problem of making it possible to combine a wide variety of component and connector types within a given system design.

In both cases, their focus is on the general-purpose nature of their languages and on providing a universal platform for architectural designs. In contrast, our research aims to exploit *architectural style* to provide more powerful support for families of systems constructed within the boundaries of that style. Thus we are willing to trade generality for power: instead of a single universal architectural development environment we promote a lot of (possibly interoperating) style-specific environments. Each such environment limits the scope of applicability, but by the same token provides new opportunities for design guidance, analysis, and synthesis.

3 What is Software Architecture and 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, GS93b, GP94]. 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 two broad categories.

Idioms and patterns: This category includes global organizational structures, such as layered systems, pipe-filter systems, client-server organizations, blackboards, etc. It also includes localized patterns, such as model-view-controller [KP88] and many other object-oriented patterns [Coa92, GHJV94].

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 [K+91], and a large variety of domain-specific approaches in areas such as avionics [BV93] and mobile robotics [SLF90, HR90].

More specifically, we observe that architectural styles typically determine four kinds of properties [AAG93]:

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 architecture [Cor91], the OSI protocol stack [McC91], and event-based tool integration [Ger89].

Fifth, as noted above, 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.

4 Automated Support for Architectural Design

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 helps developers build applications that conform to a particular Softbench event-based style [Fro89]. 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.

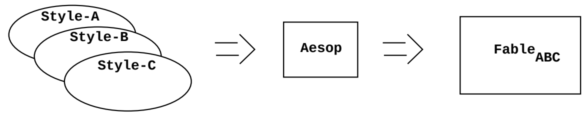

Figure 1: Generating Fables with Aesop

In order to do this, however, a number of foundational questions need to be answered: How should we represent architectural descriptions? How can we describe architectural styles so that they can be effectively exploited in an environment? How can we accommodate different styles in the same environment? How can we ensure that support for architectural development dovetails with other software development activities? In the remainder of this section we provide one set of answers to these questions.

4.1 Aesop

Aesop is a system for developing style-specific architectural development 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; and (5) multiple style-specific visualizations of architectural information together with a graphical editor for manipulating them.

Building on existing software development environment technology, Aesop adopts a "generative" approach. As illustrated in Figure 1, Aesop combines a description of a style (or set of styles) with a shared toolkit of common facilities to produce an environment, called a *Fable*, specialized to that style (or styles).

To give the flavor of the approach and to illustrate how different styles result in quite different environments, consider snapshots of three different Fables. Figure 2 illustrates the output of Aesop for the "null" style: that is, no style information is given. In this case the user can create arbitrary labelled graphs of components and connectors with the system-provided graphical editor. Both components and connectors can be described hierarchically (i.e., can themselves be represented by architectural descriptions). These descriptions are stored in a persistent object base. Additionally, the user can invoke a text editor to associate arbitrary text with any component and connector.

In terms of the four stylistic properties outlined in Section 3, the design vocabulary is generic (components, connectors, etc.), the topologies are unconstrained, there is no semantic interpretation, and the analyses are confined to topological properties – such as the existence of cycles and dangling connectors. The associated tools consist of a graphical editor and a text editor for annotations. Hence, the resulting environment provides little more than informal box-and-line descriptions, such as one might find in any number of CASE environments.

In contrast, Figure 3 shows a Fable for a pipe-filter style. In this case, the style identifies (in ways to be described later) a specific vocabulary: components are filters and connectors are pipes. Filters perform stream transformations. Pipes provide sequential delivery of data streams between filters. Topological constraints include the fact that pipes are directional, and that at most one pipe can be connected to any single "port" of a filter. Filters can be decomposed into sub-architectures, but pipes cannot. Furthermore, the environment uses the semantics of the style to provide

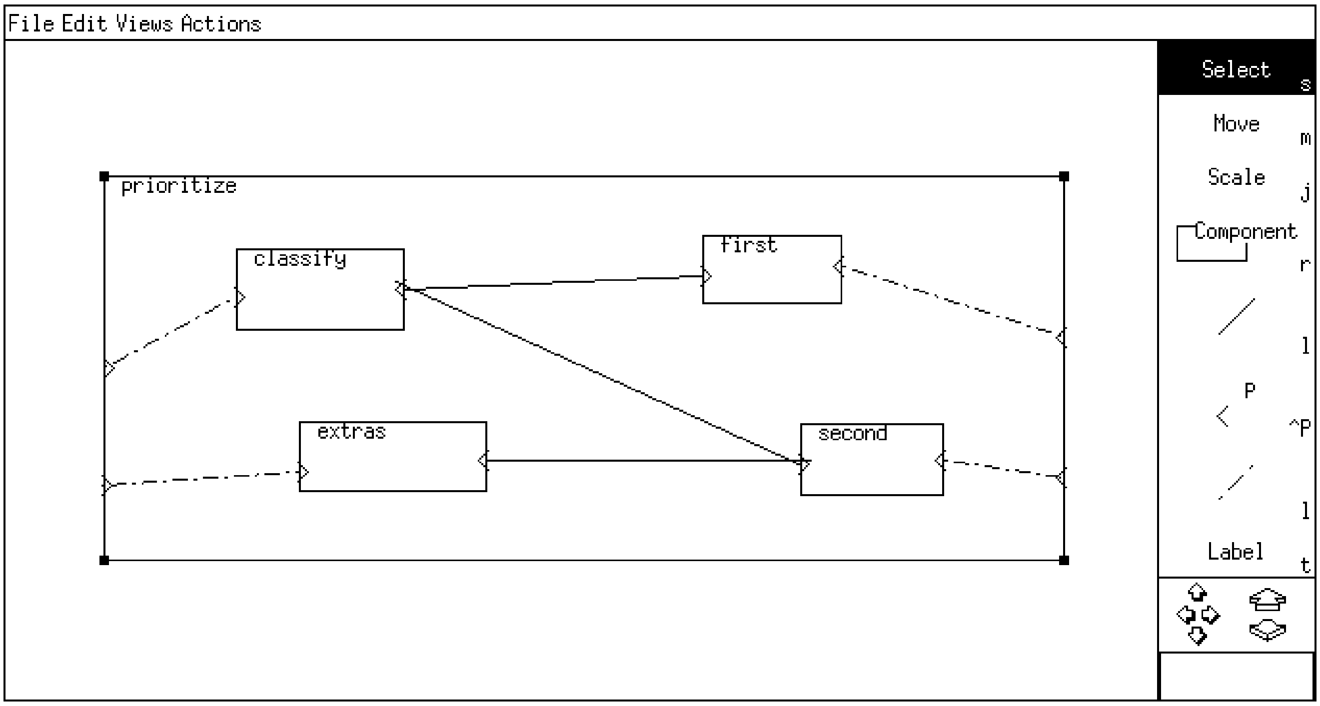

Figure 2: A "Style-less" Fable

specialized visualizations, as well as to support the development of semantically consistent system architectures. A syntax-directed editor may be used to describe the computation of individual filters. Pipes are drawn as arrows to indicate the direction of data flow. Color is used to highlight incorrectly attached pipes (not shown). Finally, the environment provides routines to check that correctly typed data is sent over the pipes, and a "build" tool uses the information present in the design database to construct the "glue code" needed to compile an executable instance of the system.

As a third example, Figure 4 illustrates an environment for an event-based style similar to Field [Rei90] or Softbench [Ger89].[1] In this environment the components are active (event-announcing) objects, and the connectors are drawn as a kind of "software bus" along which events are announced and received by the components. In this case the connector can be "opened" to expose its underlying representation as an event dispatcher. This sub-architecture is described in a different style – namely, one in which RPC is used as the main connector and the dispatcher acts as a server in a client-server style. This example illustrates heterogeneous use of styles within a single Fable. That is, the style used to represent the internal structure of a component can differ from the style in which the component appears.

With this brief overview as background, we now turn to the technical design on which Aesop is based.

4.2 The Structure of a Fable

Aesop adopts a conventional structure for its environments: a Fable is organized as a collection of tools that share data through a persistent object base (Figure 5). The object base runs as a separate server process and provides typical database facilities: transactions, concurrency control, persistence, etc. In the initial prototype the database was built by "serverizing" OBST, a public domain, C++-oriented database.[2]

Tools run as separate processes and access the object base through an RPC interface called the "Fable Abstract Machine" (or FAM), which defines operations for creating and manipulating architectural objects. This interface is defined as a set of C++ 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. Currently we use Hewlett Packard's Softbench [Ger89] for event-based tool invocation. 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 a Fable is centered around a graphical editor and database browser provided by the Aesop system. As was illustrated in the examples earlier (and explained in more detail later), this tool can be customized to provide style-specific displays and views. The current graphical editor is based on the UniDraw framework of InterViews [LVC89], a C++-based GUI toolkit. While this editor is provided as a default, it is important to note that it runs as a separate tool, and can be easily replaced or augmented with other interface tools. (For example, we have recently added an alternative interface based on Tcl/Tk [Ous94].)

4.3 Representing Architectural Designs

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. The answer to this question is critical, since,

[1] The style shown in this example is only partially implemented in our current prototype.

[2] In our most recent version, OBST has been replaced by the Exodus [C+ 90] storage manager.

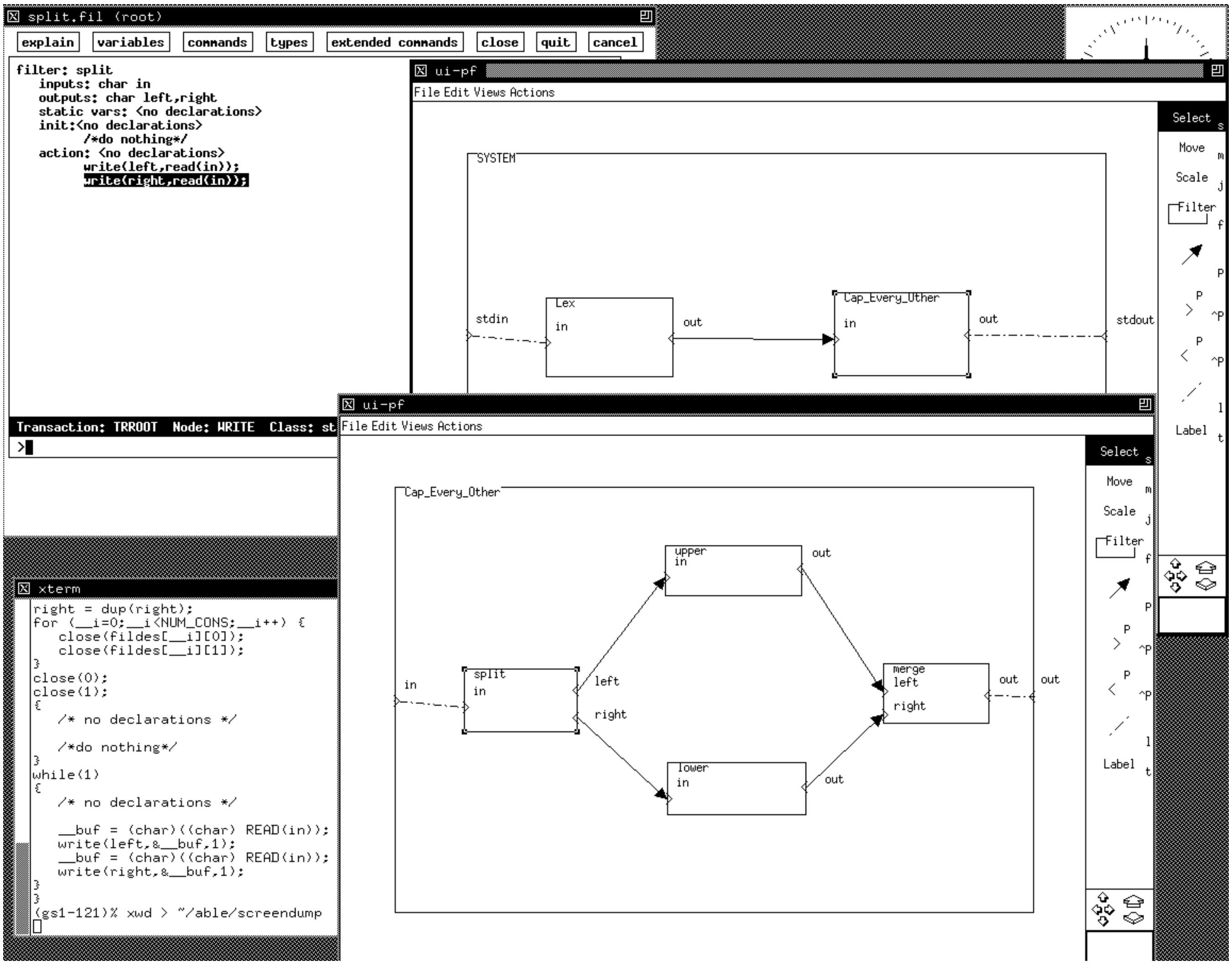

Figure 3: A Pipe-Filter Fable

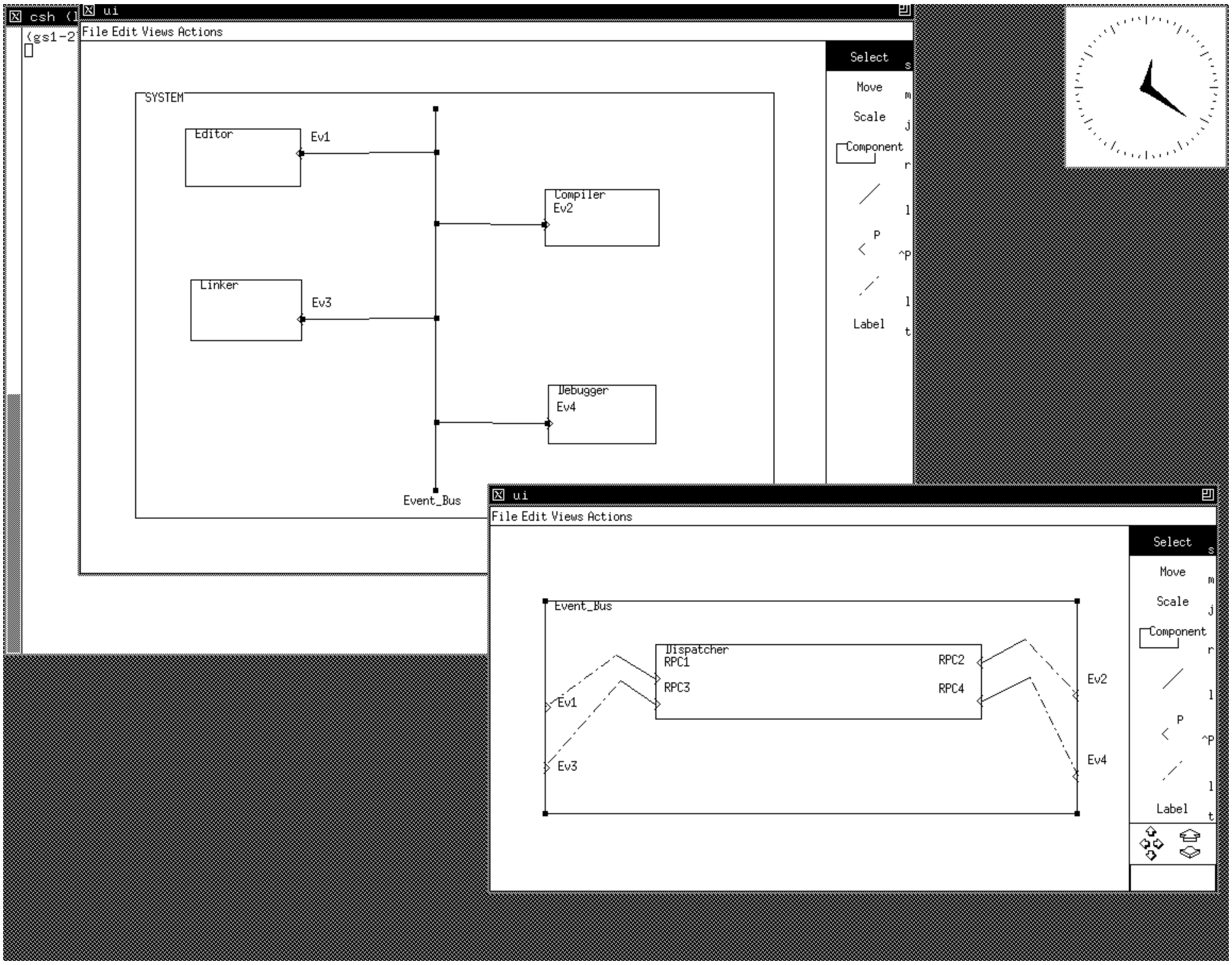

Figure 4: An Event-Based Fable

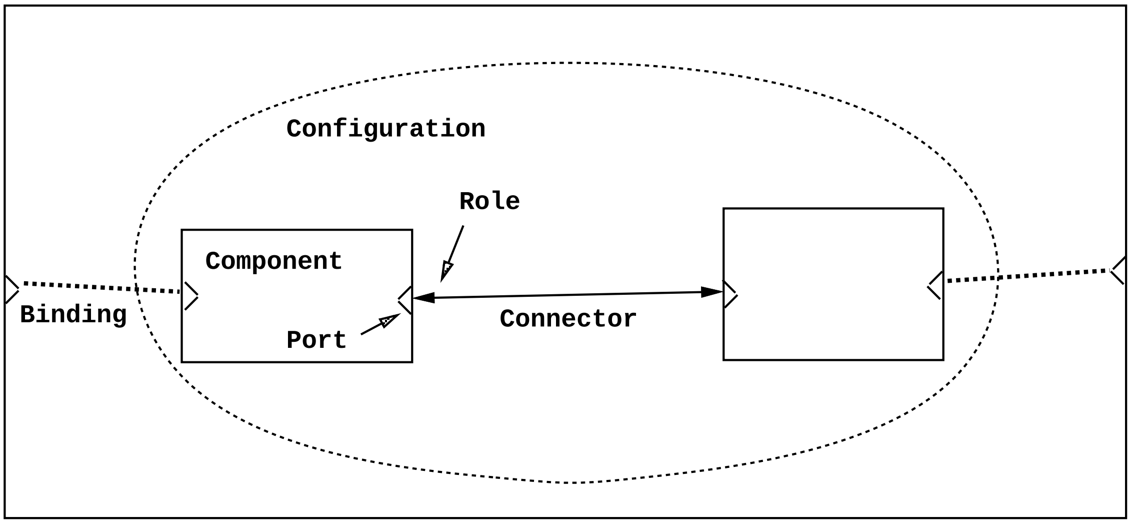

Figure 6: Generic Elements of Architectural Description

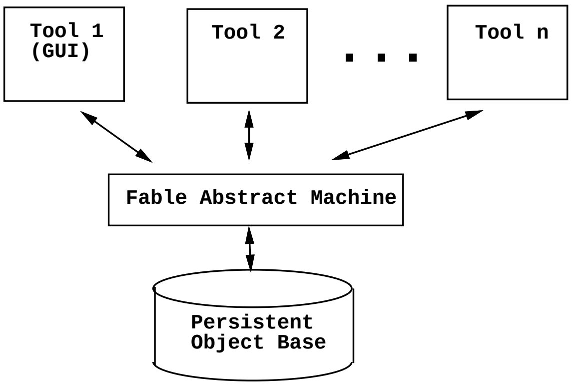

Figure 5: The Structure of a Fable

in effect, it answers the deeper question: what is an architectural design and how is it represented?

Our approach to architectural representation is based on a generic ontology of seven entities: components, connectors, configurations, ports, roles, representations, and bindings. (See Figure 6.)

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.[3]

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 example, Figures 3 and 4 illustrated architectural representations of a component and a connector (respectively).

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).[4]

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 C++ class. Operations supported by these classes include adding and removing ports to components,

[3]An argument for representation of connectors as first class semantic entities is beyond the scope of this paper, but can be found elsewhere [AG94a, Sha93]. Also, for a more formal treatment of the architectural model see [AAG93].

[4]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.

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. Collectively the classes define a Fable abstract machine interface for the null-style environment.

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 its protocol [AG94b]. 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.

Before leaving this outline of our generic object model for architectural representation, it is worth highlighting the aspects of our approach that are unusual. While the view of architecture as compositions of components and connectors appears to be gaining general acceptance, our approach has several distinctive features. First is the treatment of connectors as first class entities: they have their own interfaces (as a set of roles); they may be decomposed into sub-architectures; and they can have associated semantic descriptions.[5] This supports the conviction that a proper foundation for architecture must allow the creation of new kinds of "glue" for combining components.

Second is our treatment of representation: architectural entities can have multiple descriptions representing alternative implementations, specifications, and views. This is unlike other approaches (such as in UniCon or Rapide) where an architectural element has a single implementation that defines its "truth". In our case, truth is in the eye of the tool that uses the architectural information to derive other related artifacts (such as executables).

Third, our *generic* interface is intentionally minimal: we provide only the bare framework for architectural description, leaving additional information to be added as stylistic elaborations. This is unlike other efforts that attempt to provide a single universal style, and therefore must build in many more primitive notions (such as event patterns, particular interface specification languages, and richer vocabularies of connectors).

4.4 Defining Styles

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 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. 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.

When proposing a subtyping discipline it is important to be clear about the underlying semantic model. Specifically, in what ways can subclasses alter the behavior of their superclasses through overriding? In our system, the rule is: architectural subclasses must respect the semantic behavior of their superclasses. The term "respect", however, is used in a non-standard way. Rather than implying behavioral equivalence (as defined, for example, by Liskov and Wing [LW93]), we require that a subclass must provide strict subtyping behavior for operations that succeed, but they may introduce additional sources of failure.[6]

[5] Currently we use the Wright language [AG94b] to define the semantics of connectors as a collection of protocols.

```
fam_bool pf_source::attach(fam_port p) {

  if (!fam_is_subtype(p.fam_type(),PF_WRITE_TYPE))
  {
    return false;
  }
  else
 {
    return fam_port::attach(p);
  }
}
```

Figure 7: Code to check source role attachment

To see why this is useful (and necessary), consider the operation *addport*, which adds a port to a component. In the generic case any kind of port may be added to a component with the result that when the list of ports is requested, the new port will be a member of the result. In the case of a filter in a pipe-filter style, however, we may want to allow a port to be added to a filter only if it is an instance of one of the port types defined in the style – namely, an input or output port. It is reasonable, therefore, to cause an invocation of *addport* to fail if the parameter is not one of these two types. On the other hand, if an input or output port *is* added, then the observable effect should be the same as in the generic case. Figure 7 shows the C++ code for doing this.

To provide more concrete detail on what sorts of styles can be built and how they behave, we now provide brief descriptions of four styles. 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.

4.4.1 A Pipe-Filter Style

As indicated earlier, a pipe-filter style supports system organization based on asynchronous computations connected by dataflow.

Vocabulary. Figure 8 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. (Figure 7 shows how this constraint is enforced by a method of the new *pf_source* class. Most of the configuration rules are equally simple, although some—such as prohibiting cycles—can be considerably more complex.)

Semantic interpretation. In the prototype pipe-filter the semantics of filters is given by a simple, style-specific filter language, as was illustrated in Figure 3. The associated tool (based on Gandalf [HN86]) provides typechecking and other static analyses. The semantics of pipes is described formally (but off-line) as in [AG94b].

[6] Of course, this can not be automatically enforced for C++.

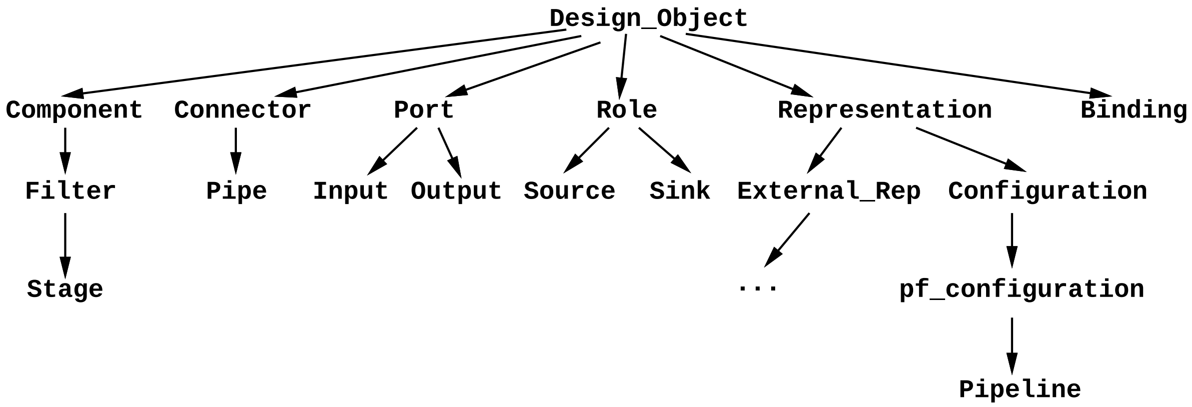

Figure 8: Style Definition as Subtyping

```
#include "filter_header.h"
void split(in,left,right)
{
char __buf;
int __i;
/* no declarations */
/* dup and close */
in = dup(in);left = dup(left);
right = dup(right);
for (__i=0;__i<NUM_CONS;__i++) {
   close(fildes[__i][0]);
   close(fildes[__i][1]);
}
close(0);
close(1);
{
   /* no declarations */
   /*do nothing*/
}
while(1)
{
   /* no declarations */

   __buf = (char)((char) READ(in));
   write(left,&__buf,1);
   __buf = (char)((char) READ(in));
   write(right,&__buf,1);
}
}
```

Figure 9: Generated code from the split filter definition

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 the external Gandalf tool. A sample of the output for the *split* filter illustrated in Figure 3 is shown in figure 9. Figure 10 shows the main body of the tool for manipulating the database to generate the executable code.

4.4.2 A Pipeline Style

A pipeline style is a simple specialization of the a pipe-filter style. It incorporates all aspects of the the pipe-filter style except that the filters are connected together in a linear order, with only one path of data flow. (This corresponds to simple pipelines built in the Unix shell.) The pipeline style is an example of stylistic sub-specialization.

Vocabulary. The pipeline style defines a new "stage" component as a subclass of filter. Its methods are identical, except that its initialization routine automatically creates a single input and output port, and the "addport" method is overridden to always fail.

Configuration rules. The configuration rules are the same as in the parent style, with the addition that the topology is constrained to be linear.

Semantic interpretation. The meaning of the pipes and filters is identical to the meaning given in the parent style. In particular, the same filter description language can be used.

Analyses. The tools of the parent style can be reused in this style, as can the code written for the parent style's classes. Since instances of subtypes can be substituted for instances of their supertypes, code written for more generic styles will continue to work on their specializations. So the compiler for the pipe-filter system will still work on pipelines. Similarly, tools developed for the null style, such as a cycle checker, will still work on instances of any of the styles in this section.

This example shows a number of benefits in using subtyping to define styles. First it provides a simple way to extend the representation and behavior of building blocks for architectural descriptions. Second, it is supported by current methodologies and tools (such as typecheckers, debuggers, and object-oriented databases). Third, it permits reuse of existing styles. New styles can be built by further subclassing of existing styles. Fourth, it allows for reuse of existing tools.

```
// Generates code for a pipe-filter system

int main(int argc,char **argv) {
  fable_init_event_system(&argc,argv,BUILD_PF); // init local event system
  fam_initialize(argc,argv);                    // init the database

  arch_db = fam_arch_db::fetch();               // get the top-level DB pointer
  t = arch_db.open(READ_TRANSACTION);           // start read transaction on it

  fam_object o = get_object_parameter(argc,argv);          // get root object

  if (!o.valid() || !o.type().is_type(pf_filter_type)) {   // not valid filter?
    cerr << argv[0] << ": invalid parameter\n";            // then stop now
    t.close();
    fam_terminate();
    exit(1);
  }

  pf_filter root = pf_filter::typed(o);
  pf_aggregate ag = find_pf_aggregate(root);           // find root's aggregate
  // (if no aggregate, print diagnostics: code omitted)

  start_main();                  // write standard start of generated main()
  outer_io(root);                // bind outer ports to stdin/out

  if (ag.valid()) {
    pipe_names(ag);              // write code to connect up pipes
    bindings(root);              // and to alias the bindings
    spawn_filters(ag);           // and to fork off the filters
  }

  finish_main();                 // write standard end of generated main()

  make_filter_header(num_pipes);  // write header file for pipe names

  t.close();                     // close transaction
  fam_terminate();               // terminate fam
  fable_main_event_loop();       // wait for termination event
  fable_finish();                // and finish
  return 0;
} // main
```

Figure 10: Main routine to generate code for pipe-filter systems

4.4.3 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 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 [Jef92]. 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. (Currently the code is targeted to Real-Time Mach.)

4.4.4 An Event-based Style

In an event-based style, components register their interest in certain kinds of events, and then can announce events and receive them according to their interest.

Vocabulary. The event style defines a new "participant" component that registers for, announces, and receives events. An "event bus" connector is used to propagate the events between components.

Configuration Rules. In this style, configuration rules simply state that the event bus connects only to components that announce or receive events.

Semantic Interpretation. Components are permitted to communicate events between each other only if they have a common bus to which they are connected, and the receiving component has registered an interest in the type of event announced by the sending component. An announced event can be received by zero or more other components (unlike in the pipe-filter style, where written data can only be read by one other component).

Analyses. A number of analyses are possible in event-based styles, such as identifying the flow of communication between components. As in the pipe-filter style, given a language for specifying the communication behavior of participant components, a compiler can be built to generate code for a particular event-based configuration [GS93a]. (We did not do this, however, in our prototype.)

4.5 User Interface

In addition to providing a representational model for tools to create and manipulate architectural descriptions, an environment must also provide a way for the user to view, edit, and use these descriptions. As we outlined earlier, the default interface is a graphical editor, which is automatically provided by Aesop and which runs as a separate tool in the environment. To produce a style-specific environment this editor (and potentially other interface tools) must also be specialized.

To accomplish this, each architectural class is associated with one or more visualization classes. New subclasses introduced by a style inherit the visualizations of their superclass, but may also define their own visualization classes. This induces a parallel hierarchy of visualization types, in which the upper portion of that hierarchy is defined by the default visualizations for the generic architectural types.

For example, to obtain the visualizations illustrated in Figure 3, the pipe subclass of connector would refer to an *arrow* visualization class, instead of the more generic *connector_line* class. Visualization classes can refer to external editors as well as to graphical objects: For example, there is a visualization class in the pipe-filter style that invokes a structure editor on filter code. The visualization classes are written in a highly stylized fashion and would be amenable to automatic generation, although we have not actually built such a tool.

Style-specific user interfaces also include object classes for user-oriented operations on the database in a particular style. These are subclasses of generic "action" classes. For instance, the user interface for a pipe-filter style may include actions to analyze the throughput of a particular configuration, or to generate code for a Unix-based implementation of the pipe-filter system. Typically these operations are carried out by external tools.

5 Evaluation and Conclusion

Aesop was developed to investigate the hypothesis that style-specific architectural development environments can be produced at relatively low cost by specializing a generic architectural model. In our research thus far we have concentrated on the important initial steps of identifying an appropriate generic model, developing mechanisms for specializing the generic model to specific styles, and providing concrete infrastructure to support architectural development tools.

While we are only now starting to apply Aesop to industrial-strength architectural styles, over the past two years we have experimented with a number of common architectural paradigms (pipe-filter, events, client-server, etc.), as well as an abstract architectural style for Aesop itself. Based on our experience thus far we are optimistic about the ability of this approach to provide useful infrastructure for the architectural level of design of software systems.

First, within the context of our prototypes we have found the generic object model for architectural representation (Section 4.3) to be an appropriate starting point for architectural description. It provides a high enough level of abstraction that it can be specialized to all architectural styles that we have yet encountered. At the same time it is concrete enough to provide both a solid conceptual structure and also associated automated mechanisms for developing new styles effectively.

Second, a subtyping model has been effective in structuring the task of developing new styles. In particular, the extension of the generic architectural model with new types provides a direct way to enrich the architectural vocabulary for design, and provide new functionality based on that design. However, as noted in Section 4.4, it is essential that the semantics of subtyping be flexible enough to allow subtypes to increase the failures associated with an inherited method.

Third, the approach is able to build on existing software environment building blocks: persistent object bases, tool integration mechanisms, and user interface toolkits. This not only provides an interface to other tools and environments based on similar technology, but has simplified the effort of building Aesop itself.

More concretely, while the costs associated with developing a style or substyle vary greatly depending on the style, typically it takes a day or two to create a minimal environment for a style with the complexity of, say, the real-time style illustrated earlier. This includes defining the new design vocabulary, encoding the constraints, and developing any new visualizations. The task of cleanly integrating the Aesop environment with existing tools that support style-specific analyses takes a bit longer. For the tools that interact only loosely with the architectural design – such as source code compilers – tool integration is little more than connecting them to our event broadcast mechanism. The hard part is adapting the tools that need to directly manipulate our database of objects, since this typically requires a deeper understanding of the tool and its implementation.

On the negative side, we discovered that there are some desired capabilities of a style-oriented architectural design environments that are difficult to handle with our approach to style definition and the technology on which Aesop is based. These capabilities fall into two categories.

The first category concerns the way in which styles are described, and includes:

- **Explicit representation of stylistic constraints.** Currently, the behavior associated with new styles – such as enforcing stylistic constraints, or enabling new kinds of analysis and tool support – must be encoded in the methods of the architectural types introduced by the style. These encodings tend to obscure the invariant properties of a style, because (a) they are bound into the imperative code of the methods, and (b) the responsibility for enforcement is often distributed over a number of different methods. This makes it difficult to reason about a style on the basis of its Aesop definition, to tell whether two styles have conflicting constraints, or to modify the policies associated with constraint enforcement and tool invocation. Approaches based on explicit rules (e.g., as in Darwin [MR88]) or inter-object mediation (e.g., as in [SN92]) are attractive alternatives.

- **Control over supertype visibility.** When a new style is defined, it is often the case that the types of design elements should be restricted to just those defined by the new style. For example, a pipe-filter style may restrict the possible port types to be only input and output ports (and therefore not allow creation of "generic" ports). This can be enforced as a constraint that is checked when a port is added to a component. But a much more natural solution would allow the style designer to restrict the accessibility of the type hierarchy.

The second category concerns the run-time behavior of style-oriented environments, and includes:

- **Dynamic incorporation of style descriptions.** In the current system our use of C++ requires us to compile style definitions at environment creation time. This precludes incorporation of new styles during execution. However, it turns out that there are many situations in which a more dynamic scheme would be useful. For example, an externally-developed repository of architectural building blocks might provide a component whose internal representation is characterized in terms of a new style.

- **Type migration.** Currently, as with most strongly typed object-oriented systems, an object's type is determined when it is created. However, it would be desirable to be able to "promote" or "demote" the type of an object at runtime. For example, if an object created as a "filter" happens to have a single input and output, it can be used as a "stage" (see Section 4.4) in a pipeline. To get the benefits of pipelines, however, we would need to change the type of the object from "filter" to "stage." While such coersions can be handled on a case-by-case basis, a more uniform mechanism, such as one based on predicate types [Cha93], would be preferable.

These features suggest ways in which style-oriented architectural design raises new challenges for software support environments. First, heterogeneity of styles is critical. Unlike software development environments centered on a single implementation language, architectural support must permit interoperability of many different design "languages".[7] Second, requirements of reusability lead to an interest in dynamic regimes for style inclusion and for types of individual design objects. Unlike most programming environments, we need to be able to introduce new types of objects and change the types of existing objects during execution. Finally, in a world of many interoperating, and independently-developed styles it is important to have good formal mechanisms for specifying new styles and for detecting conflicts between existing ones.

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 like to thank the students and staff who have contributed to the Aesop implementation described in this paper: Mike Baumann, Chanakya C. Damarla, Steven Fink, Doron Gan, Huifen Jiang, Curtis Scott, Brian Solganick, and Peter Su. Finally, we thank the anonymous referees for their constructive suggestions.

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.

[7] Referring to Section 3, the analogy between architectural style and language is based on the identification of architectural vocabulary with language syntax, structural constraints with static checks, and semantic interpretations with dynamic semantics.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to give meaning to software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 118(3), pages 9–20. ACM Press, December 1993.

[AG92] Robert Allen and David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proceedings of IFIP'92*. Elsevier Science Publishers B.V., September 1992.

[AG94a] Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, volume 29(8). SIGPLAN Notices, August 1994.

[AG94b] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994.

[B+88] G. Boudier et al. An overview of PCTE and PCTE+. In *Proc. 3rd Software Development Environments Symposium*, November 1988.

[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.

[C+90] M. Carey et al. The EXODUS extensible DBMS project: An overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

[Cha93] Craig Chambers. Predicate classes. In *Proceedings of ECOOP '93*, 1993.

[Coa92] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):153–159, 1992.

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

[DGHKL84] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The Mentor experience. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*. McGraw-Hill Book Co., 1984.

[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.

[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: Micro-Architectures for Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.

[GP94] David Garlan and Dewayne Perry. Software architecture: Practice, potential, and pitfalls. In *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994. Panel Introduction.

[GS93a] David Garlan and Curtis Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD, May 1993.

[GS93b] David Garlan and Mary Shaw. An introduction to software architecture. In V.Ambriola and G.Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume I*, New Jersey, 1993. World Scientific Publishing Company.

[HN86] A Nico. Habermann and David S. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[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.

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, AZ, December 1992.

[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.

[K+91] Rudolf K. Keller et al. User interface development and software environments: The Chiron-1 System. In *Proc. 13th International Conference on Software Engineering*, 1991.

[KFP88] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, pages 40–49, May 1988.

[KP88] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August/September 1988.

[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, to appear*, 1995.

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

[LW93] Barbara Liskov and Jeannette Wing. A new definition of the subtype relation. In *Proceedings of ECOOP '93*, July 1993.

[Mak92] Victor W. Mak. Connection: An inter-component communication paradigm for configurable distributed systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, UK, March 1992.

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

[MQR94] Mark Moriconi, Xiaolei Qian, and R. A. Riemenshneider. A formal approach to correct refinement of software architectures. Technical Report SRI-CSL-94-05, SRI International Computer Science Laboratory, April 1994.

[MR88] Naftaly H. Minsky and David Rozenshtein. A software development environment for law-governed systems. 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).

[Nii86] H. Penny Nii. Blackboard systems Parts 1 & 2. *AI Magazine*, 7 nos 3 (pp. 38-53) and 4 (pp. 62-69), 1986.

[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.

[R+91] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rei90] S.P. Reiss. Connecting tools using message passing in the Field Environment. *IEEE Software*, 7(4):57–66, July 1990.

[RT89] Tom Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Contstructing Language-Based Editors*. Springer-Verlag, 1989.

[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, to appear*, 1995.

[Sha93] Mary Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[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.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[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).

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

[vLDD+88] A. van Lamsweerrde, B. Delcourt, E. Delor, M. C. Schayes, and R. Champagne. Generic lifecycle support in the ALMA environment. *IEEE Transactions on Software Engineering*, 14(6):720–741, June 1988.